DOCUMENT RESUME

ED 182 139 SE 029 842

TITLE	Proceedings: DISE Workshop on Microprocessors and Education (Fort Collins, Colorado, August 16-18, 1976).
INSTITUTION	Pittsburgh Univ., Pa. Dept. of Electrical Engineering.
SPONS AGENCY	National Science Foundation, Washington, D.C.
PUB DATE	76
GRANT	NSF-GZ-2997
NOTE	111p.: Not available in hard copy due to marginal legibility of original document.
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	*Computers: *Electronic Data Processing; Electronic Equipment: *Electronics Industry; Engineering; Engineering Education; Higher Education; *Science Curriculum; Science Education; Technical Education
IDENTIFIERS	*Microcomputers; *Microprocessors

ABSTRACT

Papers presented during four sessions of a workshop, which addressed the role of microprocessors in education, are included in this publication. The issues covered involved seven areas: (1) views of the microelectronics industry; (2) microprocessor architecture; (3) microprocessor chip design; (4) microprocessor software; (5) the impact of microprocessors on the curriculum; (6) microprocessor laboratories; and (7) microcomputer systems and applications. (RE)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

PROCEEDINGS

DISE WORKSHOP ON MICROPROCESSORS AND EDUCATION

U.S. DEPARTMENT OF HEALTH EDUCATION & WELFARE NATIONAL INSTITUTE OF EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY.

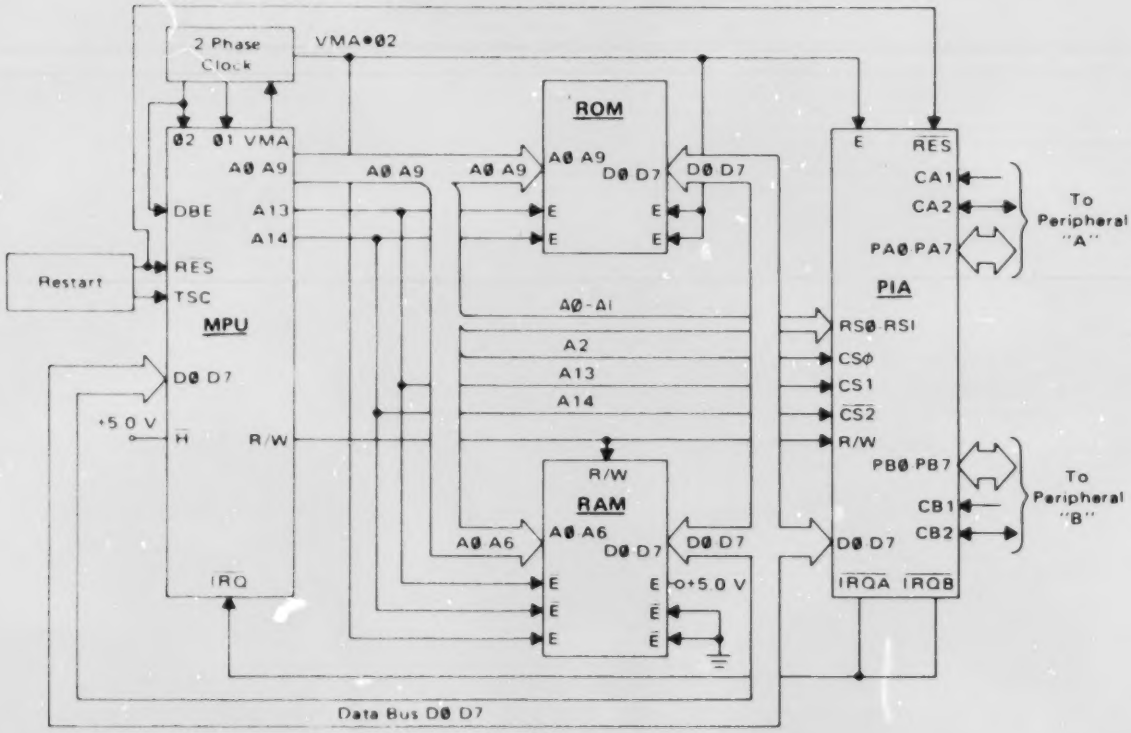
PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

Mary L. Charles
NSF

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC).

J. T. Cain, Workshop Chairman
Y. Chu, Program Chairman

ED182139



SE 029 842

AUGUST 16-18, 1976 COLORADO STATE UNIVERSITY FORT COLLINS, COLORADO

Copyright © 1976 by the
University of Pittsburgh, Pittsburgh, PA 15261

Copies can be ordered from:

DISE Project
Electrical Engineering Department
University of Pittsburgh
Pittsburgh, PA 15261

The DISE Workshop on Microprocessors and Education is sponsored by:

- Electrical Engineering and Computer Science Departments, Colorado State University, Fort Collins, Colorado.
- DISE (Digital Systems Education) Committee funded by National Science Foundation Grant No. GZ2997 and administered by the Electrical Engineering Department of the University of Pittsburgh, Pittsburgh, Pennsylvania.

The cover illustration (from M6800 Microprocessor Programming Manual) is used with permission from Motorola, Inc.

WORKSHOP ORGANIZERS:

Chairman:	J. T. Cain, University of Pittsburgh
Vice Chairman:	P. T. Lynch, National Semiconductor Corp.
Program Chairman:	Y. Chu, University of Maryland
Conference Committee:	T. A. Brubaker, Colorado State University
	R. G. Hoelzeman, University of Pittsburgh
	G. G. Langdon, IBM Research
	A. Osborne, Osborne and Associates
	R. P. Voith, The University of Toledo

INTRODUCTION

SCOPE:

This workshop is specifically intended for participation by educators involved in digital systems education. The purposes are: to present the fundamentals of and latest developments in microprocessors; and to enable educators to focus on the impact of microprocessors on curricula; and to identify what is needed to effectively use microprocessors in an educational environment.

FEATURES:

1. VIDEO-TAPES: Video-tapes will be available covering the fundamentals of microprocessors, several microprocessor architectures and the use of microprocessors in logic design. These video-tapes will be prepared by Dr. Adam Osborne and will follow his books, "An Introduction to Microcomputers" (two volumes) and "Programming for Logic Design". Copies of all three volumes will be mailed to each registrant so that they may be studied prior to attending the workshop. The video-tapes will be available each day and evening of the workshop.
2. PROTOTYPE SYSTEMS: Prototype systems and field engineers from Fairchild, Intel, National, IMS Associates, E & L Instruments, and Bringham Young University will be available each day and evening of the workshop. This is to enable workshop attendees to gain "hands on" experience with one or more of these representative systems. Nine instruction manuals from the vendors will be supplied to each workshop participant. These include Fairchild's F8 Circuit Data Book, and F8 Microprocessor Programmers Guide; Intel's 8080 Microcomputer Systems Users Manual, and 8080 Assembly Language Programming Manual; National's PACE Technical Description, and PACE Logic Designers Guide; Advanced Micro Devices Inc.'s Am 2900 Bipolar Microprocessor Family, and Am 2901, Am 2909 Technical Data; E & L Instruments' Bugbook III.

TECHNICAL PROGRAM

SESSION 1: PANEL SESSION: "VIEWS OF THE MICROELECTRONICS INDUSTRY"

Chairman: Adam Osborne, Osborne and Associates

Panelists: William Baker, National Semiconductor
J. E. Bass, Rockwell International
William Davidow, Intel Corporation

SESSION 2: "MICROPROCESSOR ARCHITECTURE"

Chairmen: E. J. McCluskey and J. F. Wakerly, Stanford University

"A Modular, Unified Microprocessor System (MUMS)" 1
M. Faiman, R. Catlin, A. Weaver, University of Illinois

"Microprogramming for the Hardware Engineer" 6
J. R. Mick, Advanced Micro Devices

"A Vehicle for Communicating the Potential Benefits of Microprocessors"
William Broderick, Intel Corporation
(Paper Not Available For Proceedings)

"Designing With A Microprocessor Prototyping Board" 15
David Eriksen, American Microsystems, Inc.

SESSION 3: "MICROPROCESSOR CHIP DESIGN"

Chairman: Ralph Ungerman, Zilog, Inc.

Organizer: Glen Langdon, IBM Research

"A Microprocessor Chip Design with the User in Mind" 18
William Wicks, Electronic Arrays

"A Course in LSI Chip Design"
Carver Mead, California Institute of Technology
(Paper Not Available For Proceedings)

"System Issues in Bipolar Chip Set Design"
Dale Mrazek, National Semiconductor
(Paper Not Available For Proceedings)

SESSION 4: "MICROPROCESSOR SOFTWARE"

Chairman: Yaohan Chu, University of Maryland

"Microprocessor Language Design" 22
Geoffrey Leach, Sycor, Inc.

"The Automated Generation of Cross-System Software for Microcomputers" 28
G. R. Johnson, R. A. Mueller, Colorado State University

"Interactive Interpreters for Microprocessor Systems" 35
Yaohan Chu, University of Maryland

SESSION 5: "THE IMPACT OF MICROPROCESSORS ON THE CURRICULUM"

Chairman: David Irwin, Auburn University

"Introduction of Microprocessor Education into an Existing Curriculum to Satisfy University and Industrial Needs" 43
Ronald Krutz, Carnegie-Mellon University

"Design with Microprocessors in an Interdisciplinary Engineering Project Course" 50
Dwight Baumann, et. al., Carnegie-Mellon University

"Microcomputers in the Computer Engineering Curriculum" 54
J. F. Wakerly, E. J. McCluskey, Stanford University

"Characteristics of a Real-Time Computer Systems Course for Engineers" 59
R. A. Volz, T. L. Boardman, University of Michigan

"Two Diverse Approaches to the Deployment of Microprocessors in a Computer Engineering Curriculum" 64
G. R. Kane, et. al., Auburn University

SESSION 6: "MICROPROCESSOR LABORATORIES"

Chairman: Peter Rony, Virginia Polytechnic Institute and State University

"Education in Microprocessors and Microcomputers" 68
L. A. Leventhal, Grossmont College

"A Microprocessor Laboratory for a University Environment" 71
Bernard J. Carey, University of Connecticut

"A Microprocessor Laboratory Course" 78
J. F. Wakerly, Stanford University

"A Mini-Computer Driven Microprocessor Laboratory for Teaching and System Development"
Wen C. Lin, Case Western University
(Paper Not Available for Proceedings)

SESSION 7: "MICROCOMPUTER SYSTEMS AND APPLICATIONS"

Chairman: Bala Parasuraman, Syscom, Inc.

"Teaching Microcomputer Interfacing to Non-Electrical Engineers" . . 84
Peter Rony, D. Larsen, Virginia Polytechnic Institute

"The Utilization of a Microprocessor Based Computer System in a Top-Down Operating System Course" 89
D. Grit, D. Georg, Colorado State University

"A Student Project in Implementing an Intelligent Computer Terminal" 93
Michael Tindall, Colorado State University

"An Undergraduate Digital System Track" 95
Allen Larson, Major, U.S. Air Force Academy

"An Interactive Open-Ended, Low-Cost, Visual Aid to Education" . . . 99
K. W. Drake, D. Salomon, San Diego State University

SESSION 8: "DISCUSSION GROUPS"

Purpose: To build on information presented in the previous seven sessions, to involve and take advantage of knowledge, expertise, and opinions of workshop participants in order to produce consensus reports of the needs of the educational community in each area. There will be four separate discussion groups, each to treat a separate topic.

GROUP 1: "What are the hardware problems and what is needed to assemble a microprocessor into a system and be effective in an educational environment?"

Chairman: Raymond P. Voith, The University of Toledo

Panelists: Gary Border, Digital Automation Associates
Larry Crum, Wright State University
Buck Jordan, Northwestern University
Imsong Lee, University of California-Berkeley

GROUP 2: "What are the software problems and what is needed to make a system work and be effective in an educational environment?"

Chairman: Gerald Johnson, Colorado State University

Panelists: David Leitch, Mostek
Al Mason, Texas Instruments
Lloyd Maul, Motorola

GROUP 3: "What are the problems in using a microprocessor as a real-time system element and what is needed for use and instruction in an educational environment?"

Chairman: Thomas Brubaker, Colorado State University

(AUDIENCE DISCUSSION SESSION)

GROUP 4: "What are the problems and what is needed to integrate microprocessors into the curriculum?"

Chairman: C. L. Coates, Purdue University

Panelists: Frederic J. Mowle, Purdue University
Edward S. Davidson, University of Illinois-Urbana
Daniel E. Atkins, University of Michigan-Ann Arbor
Bernard J. Carey, University of Connecticut

A MODULAR, UNIFIED MICROPROCESSOR SYSTEM (MUMS)

M. Faiman, R. W. Catlin and A. C. Weaver
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Summary

The would-be microprocessor user is currently confronted with a large and increasing number of different devices, each characterized by a unique architecture, instruction set and hardware conventions, and provided with a varying, usually small, degree of software support. Consequently, microprocessor systems reflect to a major extent the hardware and software idiosyncracies of the specific microprocessor (s) they incorporate; any attempt to replace a microprocessor with a newer or improved type, however inexpensive, entails a major system redesign. MUMS is an ongoing research project to overcome these limitations, having a modular, standardized and versatile structure that is nevertheless commensurate with the low cost of the microprocessors themselves. Basic to this idea is a relatively simple micro-bus that carries generic signals only, and over which a microprocessor communicates with its environment. Each microprocessor and all other modules, whether memory or I/O devices, connect to the micro-bus through simple interfaces that standardize to the bus protocol. Simplicity is retained by having one microprocessor per micro-bus, but flexibility is provided by allowing micro-busses to connect to a communications unit for the purpose of inter-processor communication or shared resources or both. By this technique MUMS can be used stand-alone or in conjunction with other processors. The insertion of a new microprocessor requires only one new micro-bus interface and an associated software utility package in a ROM module; the rest of the system is unmodified. The design of MUMS, having undergone two iterations, is approaching completion and is intended for use as a research tool and teaching aid.

1. MUMS Goals and Rationale

We are attempting in this work to take a broad view of microprocessors and their applications, to impose some structure on this field where there is very little at present, and to investigate measures of the hardware/software trade-off as it applies in the context of low cost microprocessor-based systems. Let us be more specific. First, concerning the hardware, there are about thirty different microprocessor types on the market. These range from 2-bit to 16-bit processors, on one or a few IC chips, running at a variety of clock rates and, most significantly, varying considerably in the nature and functions of the signals they receive and transmit. It is this last property that is chiefly responsible for requiring every microprocessor system to incorporate hardware that is specific to the particular microprocessor(s) it contains. Even in systems that are in no way general-purpose (e.g., intelligent terminals, peripheral controllers, computer 'hobbyist' kits, and others), the cost of the microprocessor itself is dwarfed by that of the interface circuitry. If, for example, one wishes to build a reasonably general-purpose computer around an Intel 8080--the best known of the extant 8-bit microprocessors--anywhere from 10 to 20 additional ICs will be needed to accomplish buffering, timing and status associated with the 8080 alone; moreover, every functional device associated with the microprocessor, whether some form of memory or an I/O interface, must be designed to heed the control conventions of the 8080. Adequate though this type of system may be for one-time applications, it is obviously unsatisfactory for systems that must possess the flexibility to evolve gracefully as improved hardware becomes available--a rapidly occurring process--and as new requirements or applications present themselves. The bane of the user community in both hardware and software has been special-purpose, ad hoc systems that could only be replaced in their entirety, at great cost, and never upgraded. Structured programming attempts to solve some of the software problems, while modularization has been applied at the hardware end.

Hardware modularization is, of course, not a new concept; it has been applied with varying degrees of success at the logic design and register-transfer (RT) levels, as exemplified by work at the University of Illinois, Washington University, and Carnegie-Mellon, to name a few. However, MUMS goes beyond these efforts in two important respects. In the first place MUMS is not dependent on a particular microprocessor type or a specific hardware technology. Secondly MUMS sits above the RT level, occupying a unique slot in what Bell[1], et al., have called the processor-memory-switch (PMS) level. MUMS modules are precisely processors, memory units, device interfaces, etc., that can not only be configured in a strictly functional manner according to a standardized communication protocol, but can also be readily expanded or upgraded as technology permits and as applications warrant.

If there is little standardization in current microprocessor hardware, the status of user support software is even more deplorable. There exists a smattering of simulators and cross-assemblers, usually on commercial time-sharing, for most of the devices and one or two 'stripped-down' compilers. Most user programming is done in assembly language, and such languages (even where they are adequately defined!) portray little attempt at syntax standardization and rarely incorporate any reasonable macro facility. If it is important to be able to replace hardware components with more desirable types when the need arises, it is all the more essential to be able to generate software for these upgraded systems painlessly, efficiently and rapidly.

Although microprocessors have found their major applications to date in dedicated, chiefly commercial, systems, their effective use in a university environment demands a greater degree of system flexibility that more closely resembles the traditional, general-purpose computer system. It is therefore essential to consider how the microprocessor fits into research and teaching activities in ways that complement and enhance existing facilities and practices. Again, beginning with hardware, MUMS will provide a readily configurable test bed for projects of a control nature where, for example, one may wish to make a direct cost comparison between hard-wired and microprocessor-driven controls. Such evaluations are often difficult to establish purely by simulation because of either inexact models (e.g., of the behavior of real digital circuits) or the presence of real-time constraints. Secondly, we hope by this work to establish a set of reasonable and uniform guidelines as to how, functionally speaking, future microprocessors should be designed. Thirdly, there is an obvious application for MUMS in computer science education. That software and hardware design are interrelated is a tired cliché,

but one to which mere lip service is paid in a great deal of computer science education. Now, however, the availability of low cost microprocessor systems can surely give educators no excuse for not demonstrating the marriage of hardware and software in a direct and stimulating manner. In a hands-on microporcessor course the barriers between software and hardware come down, allowing students to grapple with system problems in a top-down fashion and to study both software and hardware instances of problem solution.

In software terms, microprocessor applications are both general and specific. In general, software has developed around a single CPU exercising a variety of resources. Primary and secondary memories in such systems tend to be relatively large, with the consequence that software systems have not proved themselves immune to the dictates of Parkinson's Law. Now, suddenly, microprocessors present the community with the prospect of an extremely large number of very cheap systems. But, to attain this goal, it is obviously out of proportion to create $100,000 systems driven by a single $20 processor. Rather one must strive for some sort of cost/processing compatibility between CPU, memory and peripherals and, in so doing, examine the structural and functional constraints of the software that is appropriate to this situation. How can several relatively small and inexpensive systems, in both hardware and software terms, effectively interact so as to tackle some of the problems that have hitherto been the domain of large machines? The microprocessor is properly viewed as a resource, therefore, and in this light it suggests some specific applications. For example, microprocessors are clumsy at performing arithmetic, but can do non-numeric data manipulation and movement often as well as larger machines. This allows one to think in terms of information storage and retrieval applications (e.g., front-end processors in communications or inventory systems) where the arithmetic is trivial but such operations as searching, insertion and deletion may predominate. Again, in the areas of compilers (language translation) and operating systems, sub-tasks capable of parallel execution can be implemented in several wholly or partially dedicated microprocessor subsystems. Many of the constraints on multiprogramming through a single CPU could perhaps be eliminated by bringing to bear the parallelism of multi-microprocessors.

2. MUMS Hardware

The basic objectives of the MUMS hardware configuration have been outlined above. Its design goals are fourfold-generality, modularity, extendability and low cost--and, like most design goals, the conflicting nature of some of these entails a degree of compromise. First, as we have pointed out, it is essential to get away from the hardware idiosyncracies of specific processors, so that memory and peripherals can be configured in a way that does not require their redesign on changing microprocessors. This means establishing a uniform control protocol which, to accommodate the varying speeds of microprocessors and memories, should be essentially asynchronous. Different applications may vary considerably in their memory and peripheral needs; the system components, therefore, should be pluggable modules of appropriate functional size. System changes at the hardware level should necessitate no more effort than exchanging a few cards. These considerations alone point strongly toward a bus-oriented architecture with the bus supporting data transfer and control activities in a uniform, standardized manner. It is chiefly the goals of generality and low cost that are in greatest conflict. Historically, bus architectures have run the gamut from extreme sophistication (e.g., devices taking bus mastership under the vigilance of a dedicated bus controller, overlapping requests for bus service with execution of service cycles, incorporating priorities for hierarchical classes of bus user, allowing certain types of concurrent activity, etc.) to relative simplicity (as in schemes for instrumentation busses). Although these conflicting requirements are not always easy to quantify and resolve, some pointers may be obtained by examining the natures of typical microprocessors themselves. First, they have usually been designed by their manufacturers to control their environment, aside from generally providing facilities for interrupt and direct memory access (DMA). This has two implications: first, little is to be gained by attempting to move this control function from the microprocessor into a separate controller; secondly, each processor is likely to require its own physical bus anyway, whether or not this is apparent to the rest of the system. Secondly, most--though not all--microprocessors have instruction execution speeds that are slow relative to access times of memories in the same technology. Despite the fact that in any instance it is always possible to conceive of and implement systems that pair slow processors with fast memory or vice versa (indeed, MUMS allows this to happen), one generally observes a processor/memory speed ratio that does not vary abruptly as the technology that implements both these components advances. For these reasons, the architecture that we have chosen for MUMS is as illustrated in Figure 1. Each processor (uP) is

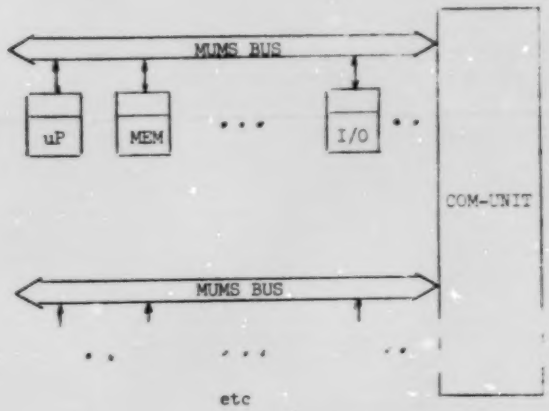


Figure 1. MUMS Hardware Architecture

allotted a standardized MUMS bus over which the processor has control. The signals on the bus are generic (i.e., functionally meaningful, as opposed to processor-specific) and allow the fundamental operations of memory read and write, priority vectored interrupt and DMA. The processor thus talks to the bus through an interface that matches the bus protocol. This interface, as it turns out, is hardly more complex than that which would be required even in a processor-specific system. All local memory and peripherals are likewise standardized to the bus conventions and here the interface design problem is in most instances simplified. For example, the handshake control for each RAM memory module requires three ICs, while that for an interrupting peripheral requires only two. Since a typical system would comprise many memory/peripheral modules but only one processor, the slight

increase in processor interface complexity is more than outweighed by the simplicity of the other modules. A single MUMS bus can therefore form the basis of a small, cheap, stand-alone system. But, in order to permit microprocessors to communicate with each other in multiprocessing applications or to share relatively expensive resources, MUMS busses can also tie into a COM-UNIT to arbitrate requests, deliver messages and generally handle the traffic flow both from one MUMS bus to another and also between the system as a whole and the outside world. The COM-UNIT will, of course, be a relatively expensive device, but again, there is only one such unit in a multiplicity of individual MUMS busses. The important point is that by partitioning the architecture in this way the most numerous modules can be implemented at the least expense.

The compromises wrought in the MUMS bus design may be illustrated by a more detailed consideration of its signals. The basic layout (Figure 2) allows

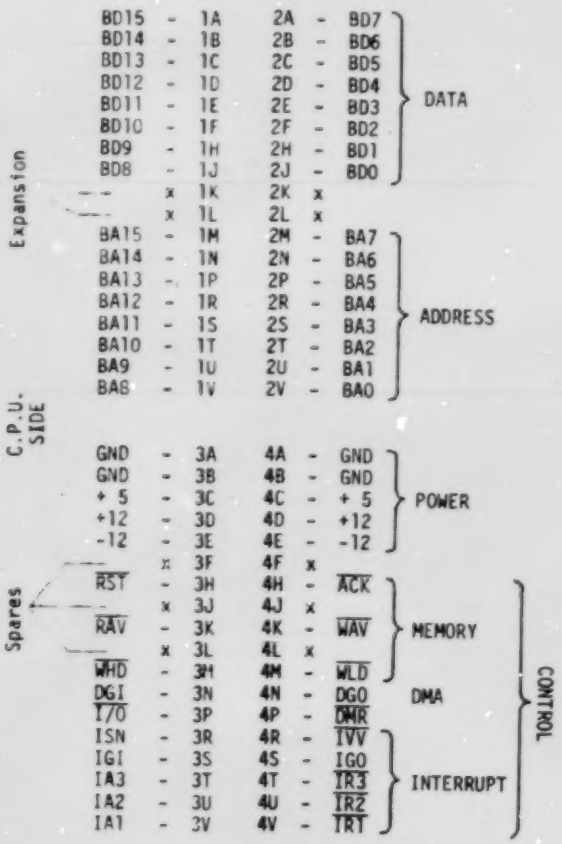

Expansion
C.P.U. SIDE
Spares
BD15 - 1A 2A - BD7
BD14 - 1B 2B - BD6
BD13 - 1C 2C - BD5
BD12 - 1D 2D - BD4
BD11 - 1E 2E - BD3
BD10 - 1F 2F - BD2
BD9 - 1H 2H - BD1
BD8 - 1J 2J - BD0
DATA
x 1K 2K x
x 1L 2L x
BA15 - 1M 2M - BA7
BA14 - 1N 2N - BA6
BA13 - 1P 2P - BA5
BA12 - 1R 2R - BA4
BA11 - 1S 2S - BA3
BA10 - 1T 2T - BA2
BA9 - 1U 2U - BA1
BA8 - 1V 2V - BA0
ADDRESS
GND - 3A 4A - GND
GND - 3B 4B - GND
+5 - 3C 4C - +5
+12 - 3D 4D - +12
-12 - 3E 4E - -12
POWER
x 3F 4F x
RST - 3H 4H - ACK
x 3J 4J x
RAV - 3K 4K - WAV
x 3L 4L x
MEMORY
WHD - 3M 4M - WLD
DGI - 3N 4N - DGO
I/O - 3P 4P - DMR
DMA
ISN - 3R 4R - IVV
IGI - 3S 4S - IGO
IA3 - 3T 4T - IR3
IA2 - 3U 4U - IR2
IA1 - 3V 4V - IR1
INTERRUPT
CONTROL

Figure 2. MUMS Bus - Signal Layout

for 16 bidirectional data lines and a 16-bit address space. Four pins located strategically between these allow for possible future expansion. The lower section of the bus is devoted to power and eighteen control signals. Of the latter the memory control signals are perhaps least open to dispute. RAV (Read Address Valid) initiates a read cycle, from either the processor or a DMA device, to which the memory or memory-mapped-input device responds with ACK (Acknowledge); ACK going off indicates data available. WAV operates similarly for the Write operation in conjunction with WLD or WHD (Write Low/High Data byte) or both, to allow for byte or word transfers. A DMA is initiated by activitating a common Request line DMR and is responded to by the processor with a daisy-chained grant DGI/DGO. Microprocessor control signals differ more significantly in the areas of input/output and interrupt handling. Memory-mapped I/O is of course possible with any processor, but to adopt this scheme exclusively would be overly restrictive for those processors incorporating explicit I/O instructions. These typically specify an 8-bit device address requiring only half the decode logic of the full address space in the device interface. Because this was considered to be a worthwhile hardware saving, to say nothing of allowing externally written I/O software to run on MUMS, an I/O line was added to the bus to allow for explicit I/O instructions. Finally, three levels of vectored priority interrupt, with level 3 the highest, were included. ISN is an Interrupt SyNch for the issuance of any request IRi, to eliminate any timing problems that the processor might otherwise have in deciding which of the IAi (Interrupt Acknowledge) signals to assert. A daisy-chained grant IGI/IGO resolves interrupts occurring at the same level, while IVV (Interrupt Vector Valid) is asserted by the granted device to indicate to the processor that its vector is on the bus.

The choice of these signals and their protocol has, we feel, provided the necessary communication flexibility at a very reasonable cost. For example, the 8080 CPU board[4], including the hardware for timing, buffering and interrupt handling, all according to the MUMS protocol, contains about twenty standard TTL packages.* However, even if this number should be regarded by some as excessive, the versatility of the chosen protocol allows the more numerous system modules, such as memory and device interface boards, to be implemented with miniscule amounts of hardware. Thus, in the typical 4k byte RAM board of Figure 3, five ICs are required for buffering and (high-order) address decoding, while a mere three accomplish the cycle timing and handshake. A similar situation prevails with the device interface modules.

At the present time, microprocessor interfaces for the 8080, LSI-11, 6502, COSMAC, F8 and GI-1600 have been designed. These are not the only processors for which we have samples in house†, but the interfaces for the remainder will be designed only as the need arises and if funding permits. The RAM and EROM modules are 4k bytes, fully buffered, with page select switches to allow a module to be placed in any 4k page of the 64k address space, and with a strappable byte-select to permit word lengths up to 16 bits. A serial I/O board also exists, as well as a simple TV character display interface which shares 1k bytes of memory with the processor. A low cost graphics system, designed for an earlier version of MUMS, is currently being modified to fit the present system. In progress, but not yet complete, are designs for a smart floppy disk controller and the COM-UNIT.

---

*Microprocessor manufacturers are continually bringing out new interface ICs. Some that can be had at the time of writing were not available when the MUMS 8080 board was designed. Were this board to be redesigned today, its functions could probably be accomplished with less hardware.

†The authors gratefully acknowledge "freebies" and cosmetic rejects received from the following manufacturers: Fairchild, General Instrument, Intel, Panafacom, RCA, Texas Instruments and Toshiba.

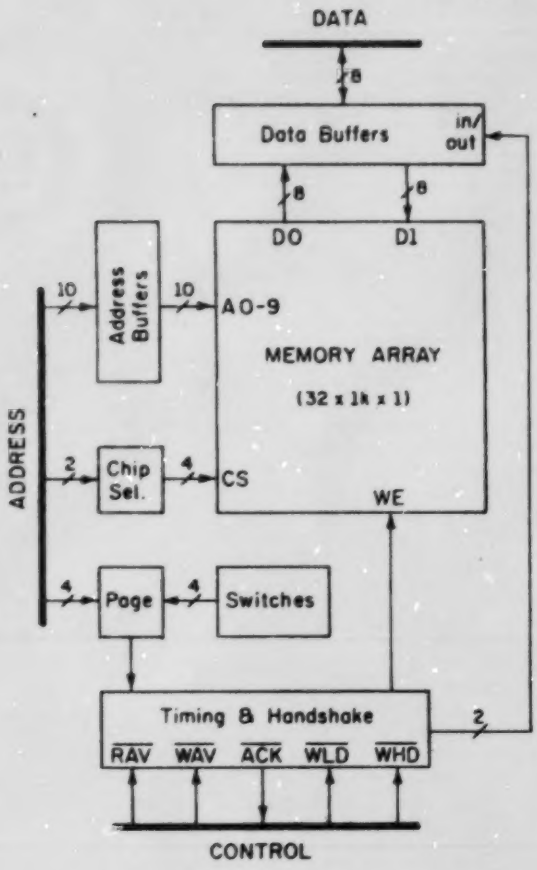


Figure 3. Block Diagram of MUMS 4k byte RAM Module

3. MUMS Software

The fundamental problem is how to impose the same qualities of structure, modularization and uniformity in the software as are found in the MUMS hardware. In dealing with one or two microprocessors one can afford to develop debug monitors, I/O drivers, cross-assemblers and even cross-compilers for each different system. But, in view of the variety of microprocessors supported by MUMS, such an approach is obviously unrealistic. Current MUMS software efforts fall into two related areas: essential software aids for bringing up new systems as rapidly as possible, and the design and implementation of an intermediate (macro) language through which machine code for any of several microprocessor types could be generated from existing compilers.

The most basic software aid is the debug monitor,[2] a processor-specific program held in a small amount (typically 256 bytes) of ROM, which is essential for program checkout when bringing up any new system. While simulators are useful for verifying program logic, they rarely simulate the vagaries of the real world. Any hardware-dependent programming where timing is critical (as in asynchronous communication schemes) is greatly enhanced by the availability of an on-line monitor. Physically, the debug monitor card and its associated I/O device interface are simply plugged into the MUMS bus whenever monitoring is desired. The monitor is interrupt driven, allows inspection and alteration of registers and memory locations and does absolute loading of programs. Communication is normally through a teletypewriter via single character (mnemonic) commands, although a parallel keyboard with CRT may be used when rapid response is needed or a large quantity of data is to be displayed. Constants may be specified in decimal, hexadecimal, octal or bi-octal (top address is 377377 — useful for 8-bit microprocessors that store addresses in two bytes) modes.

In heavy use at the present time is a "universal" cross-assembler[4] written in PASCAL and running on a PDP-10. Upon execution, this program loads internal tables and sets mode switches according to machine-dependent information contained in an external file and then proceeds to assemble the source file. Each machine file specifies how an instruction mnemonic is to be processed by means of an instruction template. A word is assembled by passing source fields through the template mask. This concept has enabled the universal cross-assembler to process source files for the 8080, COSMAC, F8, 6502 and IMP-16 microprocessors. Code for other processors can be generated by appending the appropriate machine specification tables, which may also include special template information.

Experience with the cross-assembler in handling the similarities and differences among microprocessor instruction sets has led into the second area of software work, as yet incomplete. The goals of this activity are twofold: to provide a "universal" macro-assembler using standardized syntax and mnemonics (M-Code) that would ease the task of coding for a variety of microprocessors and, secondly, to utilize M-Code as an intermediate language in effectively generating machine code from high-level (compiler) languages. This is somewhat similar to the approach taken by King[3] and Popper[5], but requires more careful design in order to optimize for characteristics of the most common microprocessor types. For the object machine description an ISP-like notation is used. A great degree of flexibility is afforded by this approach. In the case of high-level language programming, the compilation is performed on a large machine (or ultimately on a MUMS network) right down to the cross-assembly, if desired, or to the M-Code level, from which final code generation can be accomplished by a local, processor-specific MUMS assembler when one is available. At the other end of the spectrum in small stand-alone systems, users program directly in M-Code which may then be interpreted on the processor in question. The advantages of this scheme are that the particular large machine used for compilation is largely incidental, the attendant modification to an existing compiler is not great so that access to several high-level languages is possible, and the M-code generated may be processed on whichever system is most appropriate to the application.

Acknowledgment

The authors would like to thank G. L. Chesson and C. Kominczak for contributions to the software, R. M. Pleva for the 6502 interface and graphic display designs and the Department of Computer Science for its continued support.

References

[1] C. G. Bell and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, 1971.

[2]Robert W. Catlin, " MUMS: A Modular, Unified Microprocessor System," M.S. Thesis, Report No. UIUCDCS-R-76-809, Department of Computer Science, University of Illinois, 1976.

[3]Willis K. King, "A Universal Translator for Microprocessors," Proceedings of the Fourth Annual Texas Conference on Computing Systems, 1975, p. 3B 1.1 - 3B - 1.5.

[4]Charles P. Kominczak, "A Universal Cross-Assembler," M.S. Thesis, Report No. UIUCDCS-R-76-803, Department of Electrical Engineering, University of Illinois, 1976.

[5]Charles Popper, "SMAL - A Structural Macro-Assembler Language for a Microprocessor," Micros and Minis, COMPCON '74, p. 147-151.

MICROPROGRAMMING FOR THE HARDWARE ENGINEER

John R. Mick
Advanced Micro Devices

INTRODUCTION

With the advent of the Am2901 four-bit microprocessor slice and the Am2909 bipolar microprogram sequencer, the design engineer can upgrade the performance of existing systems or implement new systems taking advantage of the latest state-of-the-art technology in Low-Power Schottky integrated circuits. These devices, however, utilize a new concept in machine design not familiar to many design engineers. This technique is called microprogramming.

Basically, a microprogrammed machine is one in which a coherent sequence of microinstructions is used to execute various commands required by the machine. If the machine is a computer, each sequence of microinstructions can be made to execute a machine instruction. All of the little elemental tasks performed by the machine in executing the machine instruction are called microinstructions. The storage area for these microinstructions is usually called the microprogram memory.

A microinstruction usually has two primary parts. These are: (1) the definition and control of all elemental micro-operations to be carried out and (2) the definition and control of the address of the next microinstruction to be executed.

The definition of the various micro-operations to be carried out usually includes such things as ALU source operand selection, ALU function, ALU destination, carry control, shift control, interrupt control, data-in and data-out control, and so forth. The definition of the next microinstruction function usually includes identifying the source selection of the next microinstruction address and, in some cases, supplying the actual value of that microinstruction address.

Microprogrammed machines are usually distinguished from non-microprogrammed machines in the following manner. Older, non-microprogrammed machines implemented the control function by using combinations of gates and flip-flops connected in a somewhat random fashion in order to generate the required timing and control signals for the machine. Microprogrammed machines, on the other hand, are normally considered highly ordered and more organized with regard to the control function field. In its simplest definition, a microprogram control unit consists of the microprogram memory and the structure required to determine the address of the next microinstruction.

UNDERSTANDING THE MICROPROGRAM MEMORY

The microprogram memory is simply a N word by M bit memory used to hold the various microinstructions. Figure 1 depicts the word number definition of an N word microprogrammed memory. For an N word memory, the address locations are usually defined as location 0 through location N-1. For example, a 256-word microprogram memory will have address locations 0 through 255.

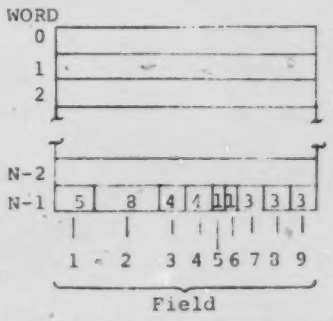


FIGURE 1: Organization of an N-Word by M-Bit Microprogram Memory

Each word of the microprogram memory consists of M bits. These M bits are usually broken into various field definitions. A typical example of field definition is shown in Figure 1 where a 32-bit word made up of nine fields is depicted. It should be noted that the fields can consist of various numbers of bits. For example, field number 1 is 5 bits wide, field number 2 is 8 bits wide, field number 6 is 1 bit wide, and so forth. It is the definition of the various fields of a microprogram word that is usually referred to as FORMATTING. Thus, Figure 1 shows the FORMAT of a 32-bit microinstruction.

An example of how microinstruction fields are defined in a typical machine microprogram memory word as shown in Figure 1 is as follows.

Field 1 - General purpose
Field 2 - Branch address
Field 3 - Next address control
Field 4 - Interrupt control
Field 5 - Fast clock/slow clock select
Field 6 - Carry control
Field 7 - ALU source operand control
Field 8 - ALU function control
Field 9 - ALU destination control

SEQUENCING THROUGH MICROINSTRUCTIONS

Once the microprogram format has been defined, it is necessary to execute sequences of these microinstructions if the machine is to perform any real function. In its simplest form, all that is required to sequence through a series of microinstructions is a microprogram address counter. Such a simplified microprogram memory address control is shown in Figure 2. The microprogram address counter simply increments by one on each clock cycle to select the address of the next microinstruction. For example, if the microprogram address counter contains address 23, the next clock cycle will increment the counter and it will select address 24. The counter will continue to increment on each clock cycle thereby selecting address 25, address 26, address 27, and so forth. If this were the only control available, the machine would not be very flexible but it would be able to execute a fixed pattern of microinstructions.

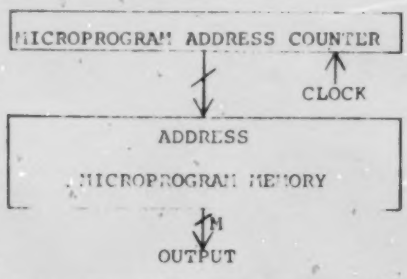


FIGURE 2: Microprogram Memory Address Control Using a Counter

The technique of continuing from one microinstruction to the next sequential microinstruction is usually referred to as CONTINUE or EXECUTE. Thus, in microprogram control definition, we will use the CONTINUE (CONT) or EXECUTE (EXEC) statement to mean simply incrementing to the next microinstruction.

MICROPROGRAM BRANCHING OR JUMPING

If the microprogram control unit is to have the ability to select other than the next microinstruction, the control unit must be able to load a BRANCH or JUMP address. Figure 3 shows a control unit architecture whereby a BRANCH address can be parallel loaded into the microprogram address counter. The load control is a single bit field within the microprogram word format. Let us call this one-bit field the microprogram address counter load enable bit. When this bit is at logic 0, a load will be inhibited and when this bit is logic 1, a load will be enabled. If the load is enabled, the BRANCH address contained within the microprogram memory will be parallel loaded into the microprogram address counter. This results in the ability to perform an N-way branch. For example, if the branch address field is eight bits wide, a BRANCH to any address in the memory space from word 0 through word 255 can be performed.

The simple branching control feature described in Figure 3 allows a microprogram memory controller to execute sequential microinstructions or perform a BRANCH (BR) or JUMP (JMP) to any address either before or after the address currently contained in the microprogram address counter.

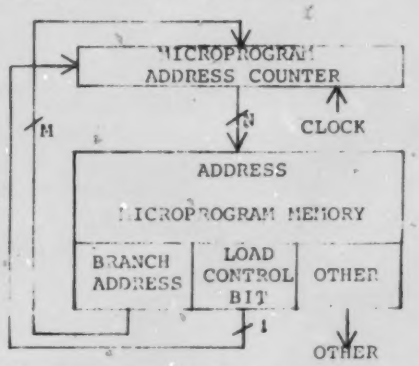


FIGURE 3: Branching or Jumping in a Microprogram Memory Space Requires a Branch Address and a Load Control Signal

CONDITIONAL BRANCHING

While the BRANCH or JUMP instruction has added some flexibility to the sequencing of microprogram instructions, the controller still lacks any decision-making capability. This decision-

making capability is provided by the CONDITIONAL BRANCH (COND BR) instruction. Figure 4 shows a functional block diagram of a microprogram memory/address controller providing the capability to branch on either of two different conditions. In this example, the load select control is a two-bit field used to control a four-input multiplexer. When the two-bit field is equivalent to binary zero, the multiplexer selects the zero input which forces the load control inactive. Thus, the CONTINUE microprogram control instruction is executed. When the two-bit load select field contains binary one, the D_1 input of the multiplexer is selected. Now, the load control is a function of the Condition 1 input. If Condition 1 is logic 0, the microprogram address counter increments and if Condition 1 is logic 1, the branch address will be parallel loaded in the next clock cycle. This operation is defined as a CONDITIONAL BRANCH. If the load select input contains binary 2, the D_2 input is selected and the same conditional function is performed with respect to the Condition 2 input. If the load select field contains binary 3, the D_3 input of the multiplexer is selected. Since the D_3 input is tied to logic HIGH, this forces the microprogram address counter to the load mode independent of anything else. Thus, the branch address is loaded into the microprogram address counter on the next clock cycle and an UNCONDITIONAL JUMP is executed. This load select control function definition is shown in Table I.

TABLE I
Load Select Control Function

S_1 S_0	Function
0 0	Continue
0 1	Jump Condition 1 True
1 0	Jump Condition 2 True
1 1	Jump Unconditional

An example of the power of CONDITIONAL BRANCHING is shown in Figures 5A and 5B. Here, ignoring the start-up problem, let us assume the microprogram address counter contains the word 0. The instruction being executed is a conditional test on Condition 1 input. If Condition 1 is true, branch to word 4 and if Condition 1 is false, continue. Word 1 performs a test on Condition 2. This test results in a branch to word 8 if true, or a continue if false. Word 2 performs a condition test on Condition 1. If true, a branch to word 12 occurs,

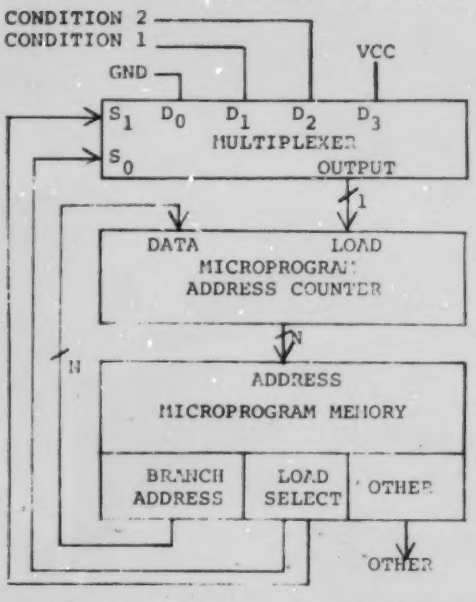


FIGURE 4: A Two-bit Control Field Can Be Used to Select CONTINUE, BRANCH, or CONDITIONAL BRANCH

MICROPROGRAM MEMORY

WORD	BRANCH ADDRESS	LOAD SELECT
0	4	1
1	8	2
2	12	1
3	0	3
4	X	0
5	X	0
6	X	0
7	0	3
8	X	0
9	X	0
10	X	0
11	0	3
12	X	0
13	X	0
14	X	0
15	0	3

X = Don't Care

FIGURE 5A: CONDITIONAL BRANCHING Example

if false a continue to word 3 occurs. Word 3 is an unconditional branch to the starting address at word 0. In this example, if a jump occurs to either word 4, 8, or 12, the net result is

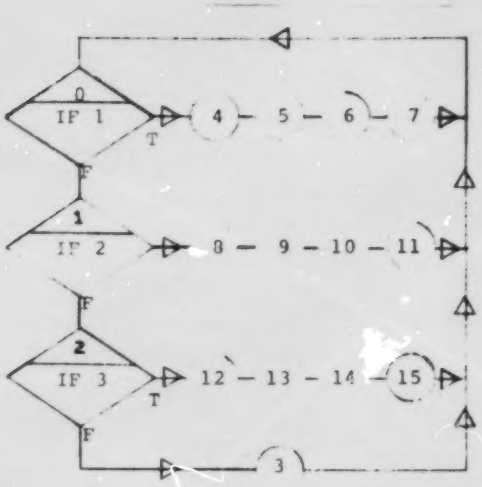


FIGURE 5B: Flow Table for CONDITIONAL BRANCH Example

that the next four instructions are executed and then an unconditional branch to word 0 occurs. The flow table for this 16-word microprogram is shown in Figure 5B.

The microprogram control flow described in Figures 5A and 5B represent a simple state machine design controller that looks for either Condition 1 to occur, Condition 2 to occur, or not Condition 2 followed by Condition 1 occurring before a reset to word 0. In the event any of the corditions do occur, a small series of tasks (four microinstructions) are to be executed and then the machine returned to word 0. If none of the test conditions occur, the machine is reset at word 3 and returned to the test at word 0. At power-up, an asynchronous reset could be applied to the microprogram address counter to initialize the machine. Although Figure 5A and Figure 5B describe an extremely simple microprogram control unit, it is representative of the microprogram control power contained in only three basic microinstructions. These microinstructions are the CONTINUE instruction, the UNCONDITIONAL JUMP instruction, and the CONDITIONAL JUMP instruction.

OVERLAPPING THE MICROPROGRAM INSTRUCTION FETCH

Now that a few basic microprogram address control instructions have been defined, let us examine the control instructions used in a microprogram control unit featuring the overlap fetching of the next microinstruction. This technique is also known as "pipelining". The block diagram for such a microprogram control unit is shown in Figure 6. The key difference when compared with previous microprogrammed architectures is the existence of the "pipeline register" at the output of the microprogram memory. By definition, The pipeline register (or microword register) contains the microinstruction currently being executed by the machine. Simultaneously, while this microinstruction is being executed, the address of the next microinstruction is applied to the microprogram memory and the contents of that memory word are being fetched and set-up at the inputs to the pipeline register. This technique of pipelining can be used to improve the performance of the microprogram control unit. This results because the contents of the microprogram memory word required for the next cycle are being fetched on an overlapping basis with the actual execution of the current microprogram word. It should be realized that when

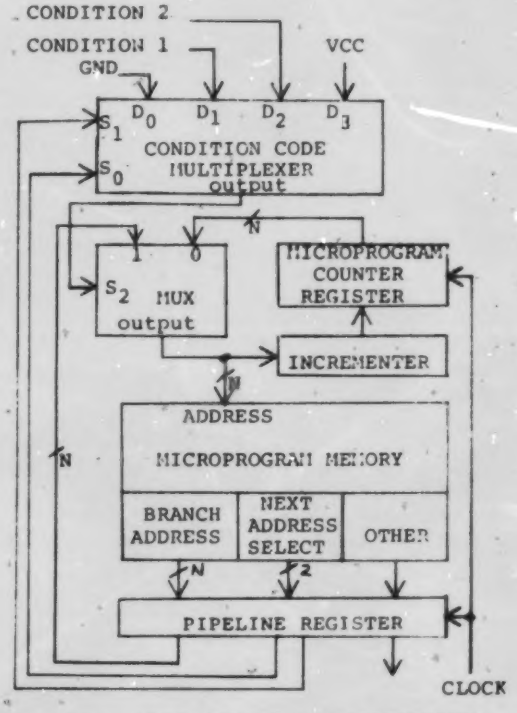


FIGURE 6: OVERLAPPING (or PIPELINING) the Fetch of the Next Microinstruction

the pipeline approach is used, the microprogram fetch, the current microinstruction being executed and the results of the previous microinstruction are available with respect to each other simultaneously. Said another way, the design engineer must be aware of the fact that some registers contain the results of the previous microinstruction executed, some registers contain the current microinstruction being executed, and some registers contain data for the next microinstruction to be executed.

Let us now compare the block diagram of Figure 6 with that shown in Figure 4. The major difference, of course, is the addition of the pipeline register at the output of the microprogram control memory. Also, notice the addition of the address multiplexer at the source of the microprogram memory address. This address multiplexer is used to select the microprogram counter register or the pipeline register as the source of the next address for the microprogram memory. The condition code multiplexer is used to control the address multiplexer in this address selection. By placing an incrementer at the output of the address multiplexer, it is possible to always generate the current microprogramming address "plus one" at the input of the microprogram counter register. In Figure 4, the microprogram address counter was described as a device such as the Am25LS161 counter. In the implementation as shown in Figure 6, the Am25LS161 counter is not appropriate. Instead, an incrementer and register are used to give the equivalent effect of a counter.

The key difference between using a true binary counter and the incrementer register described here is as follows. When the branch address from the pipeline register is selected by the multiplexer, the incrementer will combinatorially prepare that address plus one for entry into the microprogram counter register. This entry will occur on the LOW-to-HIGH transition of the clock. Thus, the microprogram counter register can always be made to contain address plus one, independent of the selection of the next microinstruction address. When the address multiplexer is switched so that the microprogram counter register is selected as the source of the microprogram memory address, the incrementer will again set-up address plus one for entry into the microprogram counter register. Thus, when the address multiplexer selects the microprogram counter register, the address multiplexer, incrementer and microprogram counter register appear to operate as a normal binary counter.

The condition code multiplexer S_0S_1 operates in exactly the same fashion as described for the condition code multiplexer of Figure 4. That is, binary zero in the pipeline register (the current microinstruction being executed) forces an unconditional selection of the microprogram register via D_0. Binary one or binary two in the next address select control bits of the pipeline register cause a conditional selection at the address multiplexer via D_1 or D_2. Thus, a CONDITIONAL BRANCH can be executed. Binary three in the next address select portion of the pipeline register causes an UNCONDITIONAL BRANCH instruction to be executed via D_3.

When the overall machine timing is studied, it will be observed that the key difference between overlap fetching and non-overlap fetching involves the propagation delay of the microprogram memory. In the non-pipelined architecture, the microprogram memory propagation delay must be added to the propagation delay of all the other elements of the machine. In the overlap fetch architecture, the propagation delay associated with the next microprogram memory address fetch is a separate loop independent of the other portion of the machine.

SUBROUTINING IN MICROPROGRAM CONTROL

Thus far, we have examined the CONTINUE instruction as well as the CONDITIONAL and UNCONDITIONAL BRANCH instructions for overlap fetch. Just as in the programming of minicomputers and microcomputers, the advantages of SUBROUTINING can be realized in microprogramming. The idea here, of course, is that the same block of microcode (or even a single microinstruction) can be shared by several microinstruction sequences of microcode. This results in the overall reduction in the total number of microprogram memory words required by the design. If we are to jump to a subroutine, what is required is the ability to store an address to which the subroutine should return when it has completed its execution. Examining the block diagram of Figure 7, we see the addition of a

push/pop stack and its associated stack pointer. The control signals required by the stack are an enable stack signal which will be used to tell the file whenever we wish to perform a push or a pop, and a push/pop control used to control the direction of the stack pointer (push or pop).

In this architecture, the stack pointer always points to the address of the last microinstruction written into the file. This allows the "next address multiplexer" to read the file at any time via port F on the next address MUX. When this selection is performed, the last word written on the stack will be the word applied to the microprogram memory. The condition code multiplexer of the previous example has also been replaced by a stack and multiplexer address control programmable read only memory (PROM). The next address control field of the microprogram word has been expanded to a four-bit field. This four-bit field in conjunction with one condition code bit form the five-bit address con-

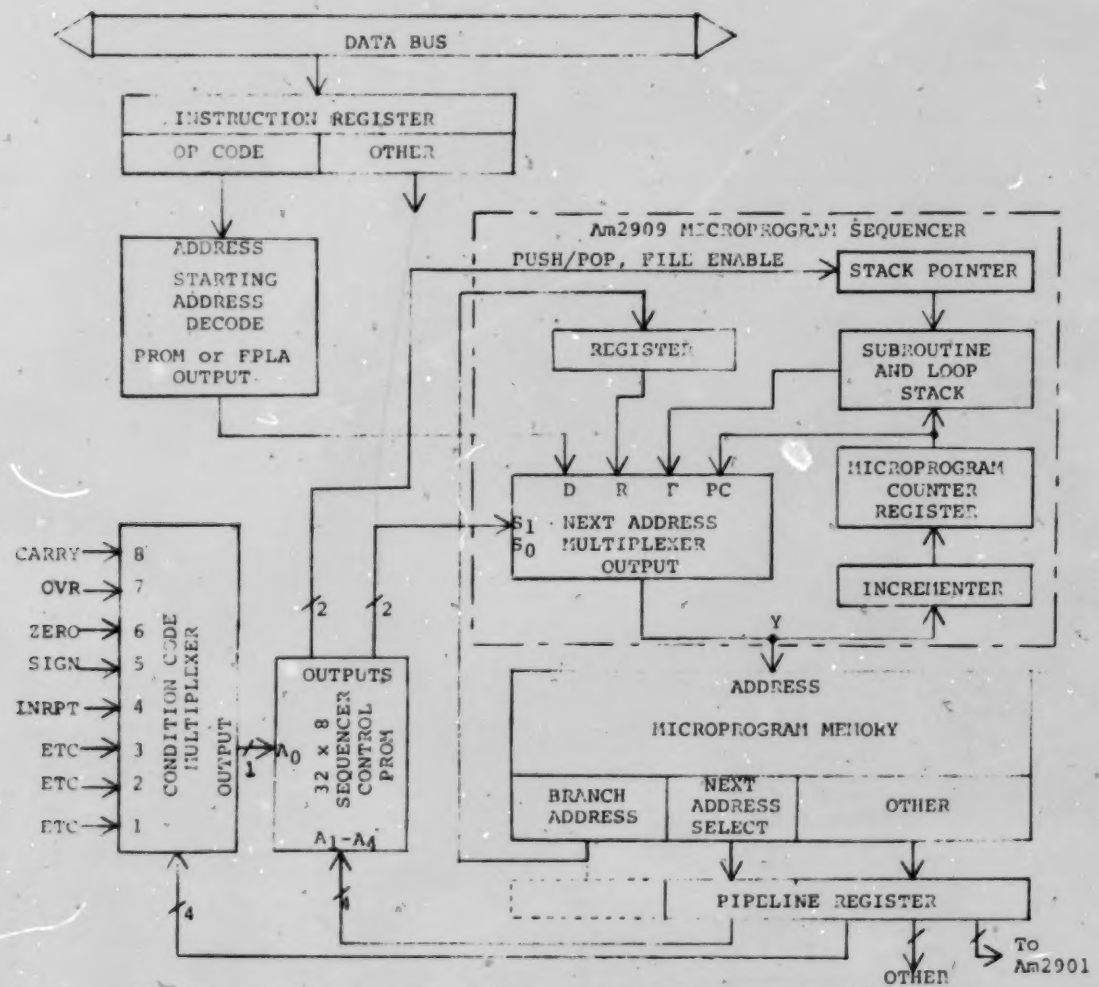


FIGURE 7: A Typical Computer Control Unit Using the Am2909

trol for the stack and address control PROM. The PROM has four outputs used to control the next address multiplexer and the stack pointer. The net result is that in this example, 16 instructions are defined where several of the instructions are conditional. Table II shows these next address select control functions.

TABLE II: SUBROUTINE CONTROL

Function	$A_4A_3A_2A_1$	A_0	Next Address MUX S_1S_0	Enable Stack	Push/ Pop
Jump to Subroutine	0 0 0 0	0	R	Yes	Push
	0 0 0 0	1	R	Yes	Push
Conditional Jump to Subroutine	0 0 0 1	0	PC	No	X
	0 0 0 1	1	R	Yes	Push
Return from Subroutine	0 0 1 0	0	F	Yes	Pop
	0 0 1 0	1	F	Yes	Pop
Conditional Return from Subroutine	0 0 1 1	0	PC	No	X
	0 0 1 1	1	F	Yes	Pop

TABLE III: PROM CONTROL FOR 16 NEXT ADDRESS FUNCTIONS

Function	$A_3A_2A_1A_0$	CC	Next Address	Stack Enable	Push/ Pop	Zero	OR
Start Address	0 0 0 0	0	D	No	X	H	L
		1					
Continue	0 0 0 1	0	PC	No	X	H	L
		1					
Branch	0 0 1 0	0	R	No	X	H	L
		1					
Jump Zero	0 0 1 1	0	X	No	X	L	X
		1					
Push	0 1 0 0	0	PC	Yes	Push	H	L
		1					
Pop	0 1 0 1	0	PC	Yes	Pop	H	L
		1					
JSB	0 1 1 0	0	R	Yes	Push	H	L
		1					
RTS	0 1 1 1	0	F	Yes	Pop	H	L
		1					
Test End of Loop	1 0 0 0	0	F	No	X	H	L
		1	PC	Yes	Pop	H	
Conditional Branch	1 0 0 1	0	PC	No	X	H	L
		1	R				
Conditional JSB	1 0 1 0	0	PC	No	X	H	L
		1	R	Yes	Push		
Conditional RTS	1 0 1 1	0	PC	No	X	H	L
		1	F	Yes	Pop		
Conditional Jump MAX	1 1 0 0	0	PC	No	X	H	L
		1	X			H	H
Jump MAX	1 1 0 1	0	X	No	X	H	H
		1					
Conditional Jump Zero	1 1 1 0	0	PC	No	X	H	L
		1	X	No	X	L	
	1 1 1 1	0	TO BE DETERMINED BY USER				
		1					

TABLE IV
Am27LS09 PROM CODING
FOR TABLE VII EXAMPLE

Address $A_4A_3A_2A_1A_0$	PROM Output $O_5O_4O_3O_2O_1O_0$
0 0 0 0 0	0 1 X 1 1
0 0 0 0 1	0 1 X 1 1
0 0 0 1 0	0 1 X 0 0
0 0 0 1 1	0 1 X 0 0
0 0 1 0 0	0 1 X 1 0 1
0 0 1 0 1	0 1 X 1 0 1
0 0 1 1 0	0 0 X 1 X X
0 0 1 1 1	0 0 X 1 X X
0 1 0 0 0	0 1 1 0 0 0
0 1 0 0 1	0 1 1 0 0 0
0 1 0 1 0	0 1 0 0 0 0
0 1 0 1 1	0 1 0 0 0 0
0 1 1 0 0	0 1 1 0 0 1
0 1 1 0 1	0 1 1 0 0 1
0 1 1 1 0	0 1 0 0 1 0
0 1 1 1 1	0 1 0 0 1 0
1 0 0 0 0	0 1 X 1 1 0
1 0 0 0 1	0 1 0 0 0 0
1 0 0 1 0	0 1 X 1 0 0
1 0 0 1 1	0 1 X 1 0 1
1 0 1 0 0	0 1 X 1 0 0
1 0 1 0 1	0 1 1 0 0 1
1 0 1 1 0	0 1 X 1 0 0
1 0 1 1 1	0 1 0 0 1 0
1 1 0 0 0	0 1 X 1 0 0
1 1 0 0 1	1 1 X 1 X X
1 1 0 1 0	1 1 X 1 X X
1 1 0 1 1	1 1 X 1 X X
1 1 1 0 0	0 1 X 1 0 0
1 1 1 0 1	0 0 X 1 X X
1 1 1 1 0	To be determined by user

0 = LOW 1 = HIGH
X = DON'T CARE

Am2909 CONNECTIONS

PROM O_0 to S_0 (Pin 16)
PROM O_1 to S_1 (Pin 17)
PROM O_2 to $\overline{FE}$ (Pin 25)
PROM O_3 to PUP (Pin 26)
PROM O_4 to Zero (Pin 15)
PROM O_5 to OR_0, OR_1, OR_2, OR_3 (Pins 12, 10, 8 and 6)

TYPICAL COMPUTER CONTROL UNIT ARCHITECTURE USING THE Am2909

The microprogram memory control described in Figure 7 is easily implemented using the Am2909. Note the use of a fourth input (D) for selecting the starting address of a sequence of microinstructions. The Am2909 thereby provides four different inputs from which the next address can be selected. These are the direct input (D), the register input (R), the program counter (PC), and the file (F).

The architecture of Figure 7 shows a macroinstruction register capable of being loaded with a machine instruction word from the data bus. The op code portion of the instruction is decoded using a mapping PROM to arrive at a starting address for the microinstruction sequence required to execute the machine instruction. When the microprogram memory address is to be the first microinstruction of the machine instruction sequence, the microsequencer control PROM selects the multiplexer D input.

The register internal to the Am2909 should be thought of as the pipeline register. When the current microinstruction being executed is selecting the next microinstruction address as a branch function, the branch address will be contained in the register internal to the Am2909. When sequencing through continuous microinstruction in microprogram memory, the program counter in the Am2909 is used. The 4 x 4 stack in the Am2909 is used for looping and subroutining in microprogram operations.

If a four-bit field in the microprogram memory format for the next address control is used for the architecture shown in Figure 7, the recommended 16 functions are shown in Table III. The 16 functions provide good control for selection of the address of the next microinstruction. The actual PROM coding for this example is shown in Table IV. The PROM selected for this implementation is an Am29705 32-word by 8-bit Programmable Read Only Memory.

FORMATTING IN MICROPROGRAM MEMORY

Many engineers designing a microprogrammed machine for the first time find difficulty in determining an approach to the microprogramming problem. At Advanced Micro Devices, we have found the following approach to be the most useful for engineers attempting their first microprogramming design job. This approach involves initially assuming only one format for the microprogram word will be used. That is, the machine architecture should be determined as required for the performance desired and the total word width of the microprogram memory temporarily ignored. This can result in a fairly wide microprogram word at the onset of the design. However, by taking this approach, the design engineer does not become immediately bogged down in making several trade-offs as to the various formats useful for the microprogram words. If the machine architecture is designed to achieve the performance required and the microprogram word is expanded such that bits are available to control the various portions of the machine as required, then the initial design task becomes straight-forward.

After the initial architecture has been laid out, the design should proceed such that small groups of microinstructions are written for the key machine instructions. Particular emphasis should be placed on the machine instruction set requiring the highest use in the firmware operation. For example, particular emphasis should be placed on the fetch routine since it is used over and over on every machine instruction cycle. Likewise, if the machine will typically execute many register-to-register arithmetic operations, the microprogram sequence used in these instructions should be reviewed carefully. As small sequences of firmware are written, it will become apparent that perhaps some fields of the microprogram could be shared in the memory. For example, it may become apparent that the interrupt control microprogram bits and the A source operand microprogram bits never occur in the same microinstruction. Thus, perhaps if four bits are used for interrupt control and four bits are used for A source operand select, then these four bits might use the same four-bit field in the memory. This would save a four-bit field throughout the entire microprogram memory addressing space. However, it will be necessary to determine which field is present in the word in the pipeline register. Thus, one bit may be dedicated in the entire microprogram word field to distinguish between format number 1 and format number 2. When this bit is a zero, the machine recog-

nizes format 1 and applies these four bits to the source operand control. When this bit is a logic 1, the machine recognizes that format number 2 is in the format register and applies this field to the interrupt control.

From this discussion, it should be apparent that perhaps four different formats might be utilized by the machine and a two-bit field used to identify which of the formats is in the pipeline register. In some designs, parts of the pipeline register may be repeated more than one time for some control fields. If this is the case, then it becomes necessary to select the appropriate register to be loaded by the microinstruction coming from the memory for this field. Thus, the two bits determining the format may be decoded using a two-line to four-line decoder such as the Am25LS139. The appropriate output from the Am25LS139 can enable a register with a clock enable function as on the Am25LS07 (four-bit), Am25LS08 (six-bit), or Am25LS377 (eight-bit) such that the microprogram memory word is loaded into the appropriate register for use by the machine.

The primary advantage of formatting the microprogram word is to save bits in width for the microprogram memory. Occasionally, as the formatting becomes more complex, it will be found that two fields are required simultaneously. Thus, this can result in an extra microinstruction being required to be able to perform all of the functions required by the microprogram control. Normally, anywhere from two to eight different formats can be found useful in the microprogramming of many machines.

For the design engineer doing his first microprogramming job, the emphasis should be on layout of the architecture of the machine to meet the required specification. Should the design accidentally use too many bits in microword width, it would be unfortunate but not catastrophic. The machine would still perform properly and meet specification and, in many cases, only one or two extra integrated circuits would have been used. Based on the cost of Field Programmable Read Only Memories, this cost is not unacceptable. Also, it is possible that such an approach may considerably reduce the overall design time of the machine. As the designer gains experience, he will become more proficient in formatting microprogram words and will soon find he can generate microprogram control architectures with reasonable speed.

SUMMARY

The Am2909 provides a unique solution to the microprogram memory sequence control problem. It is particularly well suited for high-performance computer control units using overlap fetch of the next address in microprogram memory. By using a microsequencer control PROM, the functional control of the next address selection can be uniquely defined. In addition, operations involving the internal stack can be selected as desired. The Am2909 is also a useful device in state machine control design. That is, control of the microprogram memory can be programmed as needed for the design.

DESIGNING WITH A MICROPROCESSOR PROTOTYPING BOARD

David K. Ericksen
American Microsystems, Inc.
Santa Clara, CA 95051

Practically every microprocessor manufacturer has introduced a prototyping board of some sort to give the potential customer a practical evaluation tool. Prototyping, or evaluation boards, come in all sizes, shapes and, as expected, each offers several alternatives. Many users are asking, "What can I do with it, how do I use it, what else must I have to make it work?" This paper discusses the purposes of a prototyping board and gives a practical problem showing how a prototyping board may be applied to solve the problem.

Purposes of Prototyping Boards

There are three general purposes for vendor prototyping boards:

1. Evaluate vendor's parts.
2. Develop and run programs.
3. Debug hardware design (assist).

Prototyping boards are priced in order that a user may purchase a small variety and use them to evaluate and compare microprocessor performances. The board ideally contains several components which support the microprocessor so that the user can examine how an entire family of microprocessor parts perform.

Program development has become a point of considerable scrutiny as more and more development dollars have gone to programming. The ability to execute user programs for debugging purposes is an absolute necessity of prototyping systems. A program debug/display capability is also essential for every microprocessor development task.

Since microprocessors must interface with a host of input/output devices, a means for proving these interfaces early in development cycles is very attractive. Prototyping boards may offer the flexibility to communicate directly or indirectly with several types of I/O devices.

What Else is Required?

Besides a prototyping board, what else does a user need to make the board work?

Power requirements may vary greatly. Generally, the voltages required for the microprocessor are all that is needed for the whole board. However, specfic memory types may require special considerations. Current usage is dependent on the amount of memory and circuits supplied with the board. Refer to the vendor's specifications for complete details.

A teletype is usually specified for input and output to the board. It is advisable to look for I/O that is flexible enough to accept any standard terminal and baud rate which may be available. Most operating systems will specify one type of I/O device.

What Can The Program Do?

The operating system program provides a convenient means of operating the MPU from the I/O device. Necessary functions include display and alter RAM, load and punch paper tape and the GO TO (exist monitor) commands. Other commands such as display registers, move memory blocks and zero memory are handy additions. These subroutines assume data handling tasks, such as ASCII/hex conversions and control character identification. The program must initialize the programmable I/O ports (if available) and do any pre-assignments to stack and index registers as required. Functional messages for I/O errors, such as in memory range (BAD ERROR) or check-sum errors, from the tape reader, are also very desireable.

Application to a Traffic Controller Design Problem

The individual components of a prototyping board can best be evaluated by discussing its use in solving a design problem. Prototyping boards should include all the components necessary to operate the microprocessor and perform a variety of I/O functions. To better explain this concept, a traffic controller intersection is discussed. (See Figure 1 for a traffic intersection diagram.)

The operation of a traffic control situation is graphically shown in Figure 2, a state diagram of an intersection, and in Table I.

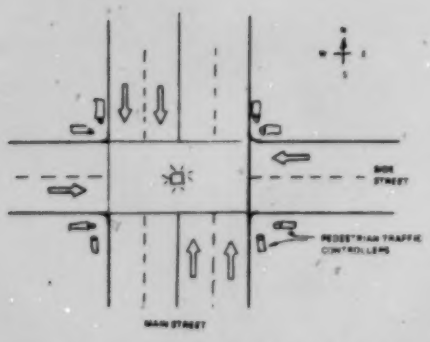


FIGURE 1 TRAFFIC INTERSECTION DIAGRAM

TABLE I TRAFFIC INTERSECTION

STATE DIAGRAM DEFINITIONS

STATE	SIGNAL STATUS	TIMES
0	Main = Green/Side = Red	Main traffic, no pedestrians
1	Main = Amber/Side = Red	Switching from Main to Side
2	Main = Red/Side = Green	Side traffic, no pedestrians
3	Main = Red/Side = Green / walk-on	Side pedestrians
4	Main = Red/Side = Green / walk-on	Side traffic, main traffic waiting
5	Main = Red/Side = Amber	Switching from Side to Main
6	Main = Green/Side = Red	Main traffic, no pedestrians
7	Main = Green/Side = Red / walk-on	Main pedestrians
8	Main = Green/Side = Red / walk-on	Main traffic, side waiting

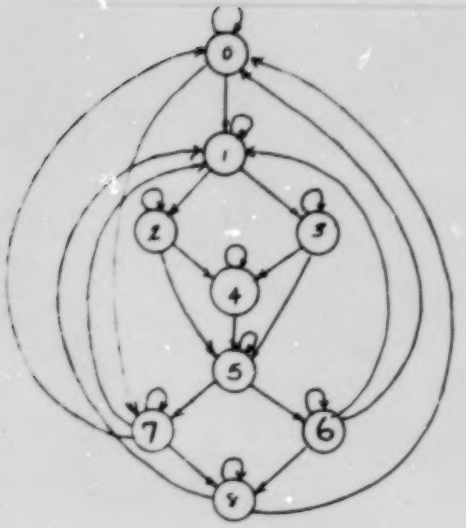


FIGURE 2 STATE DIAGRAM FOR TRAFFIC INTERSECTION

To apply a prototyping board to this problem, there are three parts which can be identified, (1) processor or controller design, (2) interface design and (3) software design. How can a prototyping board assist in each of these areas?

Controller Design

The board may provide a model for clock, reset and address decode circuits. If modeled after the same address space, the board could then be used to test the controller design. (Use the RAM on the board for testing design.) The functions and characteristics of the processor under design should be displayable from the board. For example, in the traffic problem, such things as controller, reset (restart), clock, bus and I/O and memory interfaces can all be tested from the board.

Interface Design

Parallel with the controller design is the design of the actual interface logic. The switch and relay and the sensor/light devices need to be modeled through a sample of actual I/O handling devices. Table II demonstrates how these may be assigned to an I/O port of a prototyping board.

TABLE II MODELING LIGHT AND SENSOR FUNCTIONS FOR A TRAFFIC INTERSECTION

INPUT/OUTPUT LINE		
INPUT	1	Northbound Traffic
	2	Southbound Traffic
	3	Eastbound Traffic
	4	Westbound Traffic
	5	Main Pedestrian Buttons
	6	Side Pedestrian Buttons
OUTPUT	1	Main Green Light
	2	Main Amber Light
	3	Main Red Light
	4	Side Green Light
	5	Side Amber Light
	6	Side Red Light
	7	Main "Walk" Lights
	8	Main "Wait" Lights
	9	Side "Walk" Lights
	10	Side "Wait" LIghts

Software Testing

Since considerable design effort allows the program to operate, these are several things a proto board must do to support this effort. Assume the design may be flow charted as show in Figure 3.

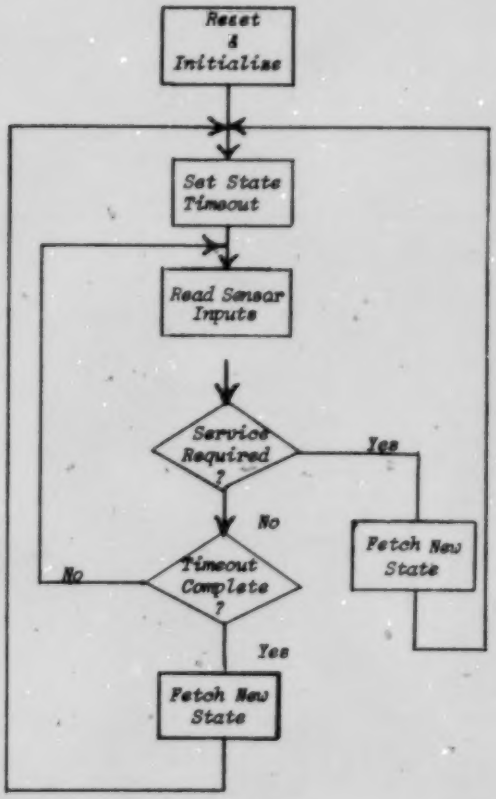


FIGURE 3 SIMPLE FLOW CHART FOR TRAFFIC INTERSECTION

The first step of coding is to get the program into the RAM of the proto board. This can be accomplished by using several methods including TTY paper tape from a larger assembly program, acoustic (Modem) couple from a fast CPU, and direct hex absolute from the TTY or terminal keyboard. Once the program is in RAM, it can be transferred to TTY paper tape for preservation and easy re-entry. With the program entered, it can then be tested using the debug program in the prototyping system and the associated interfaces.

System Development

The development steps which may be taken to implement this system are shown in Table III along with a reference to the utilization of the proto board.

DEVELOPMENT STEP	HARDWARE DESIGN	SOFTWARE DESIGN
Specify	Test & Familiarize with Proto Board I/O Interfaces	Program Functional Familiarization
Design	Model Hardware Design from Prototype Board Circuits	Assign Memory and I/O Addresses similar to Prototype Board
Simulate	Test Interface Circuits through Prototype Board I/O	Run Program Model on Prototype Board
Test	Test completed designs Prototype Board as a board tester	Generate new test program for diagnostics and Final Test
Support	Inspect Incoming parts, Model any EC activity	Model any Program Changes

TABLE III DESIGN STEPS USING PROTOTYPING BOARDS

As the design is prototyped, the board may interface with memory modules, model the program and interface with I/O circuits. A good design may allow usage for final test production devices when they are assembled. A complete development assistive unit provides an EPROM programming capability.

The total usage of a proto board includes each phase of a problem development. Familiarization and use of these development tools offer many hours of savings in the development cycle.

References

AMI Hardware Reference Manual: Chapter 6, page 101, "An Example System," and AMI Prototyping Board Applications Manual.

Contributing to the publication of the manuals were Ralph Bestock, J.R. Davis, D.K. Erickson, and T. Pittman.

A Microprocessor Chip Designed With The User In Mind

By W. E. Wickes
Manager, Microprocessor Dev.
Electronic Arrays, Inc.

Electronic Arrays is producing the EA9002, a new single chip, 8-bit parallel microprocessor. If you are close to this market and familiar with most of the available or nearly available microprocessors your first reaction should be: "Another one - Why?"

This paper is not so much on WHAT is the EA9002, but WHY it was created. In this way, a "real world" flavor to the why of microprocessor chip design can be imparted.

A microprocessor is made by process engineers using information from circuit, logic, software, computer and systems engineers. A microprocessor is used by logic, software, computer and systems engineers. Any good microprocessor product must be the result of trade-offs across this broad spectrum of technologies. But, these trade-offs must be toward the solution of a pre-defined problem. Without this prior study, a microprocessor will look like it was first designed as a solution and then went looking for problems to solve.

Okay, what is the problem? EA's analysis indicates that the highest potential volume of microprocessor users will not be the typical computer data processing products but rather the small process controllers, smart instruments, data collection terminals, specialized peripheral controllers, video games, etc. That is, products which are not large "number crunchers," but digital data processors and controllers. Products which do not require 16K or 32K bytes of memory but rather 1K to 4K bytes of memory. Products with speed requirements ranging from very slow to very fast. Low cost products - $50 to $5000. Requiring minimum component count at the systems level. Flexible - able to interface with a broad spectrum of peripheral components from standard TTL SSI devices to A - D and D- A converters to smart communications modem controllers.

There is no such thing as a general purpose microprocessor which is cost effective in all applications. Any given microprocessor can be shown to be more cost effective than another in a specific application. Just as a given minicomputer can be more effective in one specific application than another. Just as one high-level software language is more effective for a specific class of problems than another. Of course, this is exactly why there is more than one high level programming language, a variety of minicomputers, and a spectrum of microprocessors. Microprocessors are forecast as a multibillion dollar market. It stands to reason that, as this current market grows, a continuous stream of new, better, faster and specialized products will emerge to satisfy this gluttonous demand.

The EA9002 is one of the first of a new breed of specialized general purpose microprocessor products emboding the very latest in architectural concepts, data handling features and semiconductor process technologies.

"Designed With The User In Mind" - just who is the user? He is the logic and hardware engineer currently working with TTL SSI and MSI components. Not the computer scientist worrying about financial and inventory control systems. He has a data collection terminal to design, an elevator to control, a video game to configure; a card reader product, a smart laboratory instrument, a remote site alarm function generator, an automated chicken plucker to implement. Above all - he is cost conscious.

Therefore, the ground rules for development of the EA9002 were established as:

a) Single +5 Volt Supply

b) Stay within 28 Pin Package
c) TTL Compatible on Inputs and Outputs
d) Simple Clock Input
e) Keep it below 200 Sq Mils
f) Make it High Speed
g) 8-Bit Parallel Architecture
h) Improve on weaknesses of current products
i) KEEP IT SIMPLE

Did we meet these objectives -- let's take a look.

SINGLE +5 VOLT SUPPLY - YES

A single +5 volt supply for MOS requires an ion implanted N-channel process. EA has this process and was able to design and implement the EA9002 with this process.

STAY WITHIN 28 PIN PACKAGE - YES

The 28 pin package was chosen primarily for cost considerations. 28 pins being considerably cheaper than 40 pins. This automatically constrained other definitions - i.e. - size of address bus, number and features of control bus and clock requirements. An on-board clock oscillator would have required two pins. It was decided to utilize an off-chip single phase oscillator requiring only one input to the MPU chip and generate all additional phases within the chip.

A RESET input was a must as was an INTERRUPT input. In order to provide the ability to interface with as broad a variety of external RAM, ROM and I/O devices as possible it was decided to generate separate strobes controlling data and instruction transfers. Thus, a $\overline{\text{DIS}}$ (Data In Strobe) and a $\overline{\text{DOS}}$ (Data Out Strobe) were defined. A final control pin $\overline{\text{WAS}}$ (Wait/Sync) was defined to enable control of the MPU when interfacing with slow external devices or for single step operation. This pin was also defined to output a negative going sync pulse at the end of each instruction execution time for program debugging and systems synchronization.

Since an 8-bit data bus was dictated, this left exactly 12 pins for the address bus. Is 12 pins, or 4096 external memory and I/O locations sufficient? Referring to the user and expected applications - yes. Furthermore, direct address space can be easily expanded by means of memory bank select techniques such that this was not considered a significant liability.

TTL COMPATIBLE ON INPUTS AND OUTPUTS - YES

This requires large MOS interface circuits and careful circuit design techniques available within EA's design and processing skills. The trade-off is in chip area - i.e. - providing space on the silicon for drivers capable of sinking 2ma current and switching at the required speed.

SIMPLE CLOCK INPUT - YES

As previously discussed, an on-board oscillator was a trade-off in pins available and silicon area required within the MPU chip. Moving the oscillator off the chip reduced chip area and provided full flexibility to the user in oscillator implementation. By generating required multi-phases on the MPU chip, the user need only implement a simple single phase clock generating circuit.

KEEP IT BELOW 200 SQUARE MILS - ALMOST

The size of a silicon chip is very cost sensitive, particularly in the range of 150 to 200 sq mils. Manufacturing cost of silicon chips is a function of yield which relates to the number of good chips realized out of the total chips possible for a given wafer size. There are approximately 150 possible 200 sq mil chips on a 3" wafer. If the chip size were 180 sq mils, there would be approximately 190 possible chips. If yield is assumed constant at 20% (not necessarily the case) then the number of good 200 sq mil chips per wafer would be 30 while for the 180 sq mil chip the yield would be 38. This may seem trivial but if you are processing 1000 wafers per month the number of good chips would

go from 30,000 to 38,000! (You wondered why deliveries fluctuate!) Compounding this is the fact that yield is very much a function of chip size. Therefore, careful study and agonizing trade-offs are made on each item which may affect chip size. For example, there are over 5,000 transistors contained within 1200 logic equations and 448 bits of static RAM. The subroutine stack consisting of eight 12-bit registers and the eight 12-bit general purpose registers are both contained within one RAM block. This implementation helped reduce chip size by minimizing the number of flip-flop registers. The instruction decode control array was implemented as a dynamic logic gating array as opposed to a PLA structure. This is somewhat less flexible than a PLA but required less space than either a PLA or ROM structure for generating macro-type control signals. The EA9002 is 200 x 201 mils. The power supply, speed, amount of logic, number of transistors and features contained within this silicon area is quite an achievement.

MAKE IT HIGH SPEED - YES

The EA9002 operates with a 4MHz input clock to achieve a 2μsec instruction fetch and execution time. This high speed, along with an efficient instruction set make it one of the fastest MPUs available. High speed, small chip size, and single +5V supply have been achieved through the use of ion implanted, depletion mode, silicon gate N-MOS semiconductor circuit design and processing technology. The same technology employed in today's high speed dynamic RAMs. Circuit speed is a very dominant factor in resulting chip size since it directly impacts internal transistor sizes. A slower circuit can be implemented in a smaller chip size. The internal propagation delay of logic elements within the EA9002 is typically 15nsec. A single instruction is addressed and fetched by the MPU within 1μsec. This instruction is executed and data transfer, if required, takes place during the next 1μsec interval. The EA9002 could have been made slower and thus smaller. To fulfill the need of the broad spectrum of potential users it was felt that "faster-is-better." The chip can always be run slower than its maximum speed but it can never be run faster.

8-BIT PARALLEL ARCHITECTURE - YES

Today's microprocessors, having started with 4-bit PMOS designs have moved rapidly to 8-bit NMOS designs. There is some demand for 16-bit architecture as evidenced by the 16-bit designs beginning to emerge. Debate centers around data processing speed, efficiency of memory utilization, and the inevitable machine language problem. Not only is "faster-is-better" but you would be led to believe that "wider-is-better." These two requirements often conflict as evidenced by those designs which approach the speed problem by bit-slice techniques. That is, multi-chip bit-slice architectures employing bipolar processing techniques to achieve sub microsecond cycle times. Mathematical routines can be handled with 4-bit decimal algorithm as is done in calculator products. However, alpha-numeric and bit data handling is best done with 8-bit organizations. Further 8-bit architectures perform packed BCD arithmetic very efficiently. Market surveys also indicate a preponderance of demand for 8-bit designs. Thus, the decision was made in favor of an 8-bit architecture.

IMPROVE ON WEAKNESSES OF CURRENT PRODUCTS - YES

The EA9002 was initiated at a time when the marketplace had had time to react to features of various microprocessors. It is easier to critique a given design than it is to conceive one. A study was made of the good and bad features being reported upon and the good were traded-off in both architecture and instruction set considerations. It can be said the EA9002 has a built-in stack for subroutine return address like the 4004, an on-board 64 byte scratch RAM like the F-8, simplified timing concepts of the PPS-4,

straight forward peripheral addressing techniques of the 6800, general purpose registers like the 8080, a unique simplified instruction set, a single phase clock and single +5 volt supply. The major strengths of the EA9002 lie in its speed, efficient one chip architecture, 64 byte on-board RAM, powerful instruction set and TTL compatibility. I don't admit to any weaknesses.

KEEP IT SIMPLE - YES

What is simple to one person may be complex and confusing to another. However, no one has ever said that microprocessors are simple. They are as sophisticated a product as has ever been conceived by man. Witness the many seminars, training courses, documentation, articles, training and development hardware aids available. Entrepreneur consultants have emerged who can "solve" your problem with a microprocessor. There are a few microprocessor products which can be assessed rather easily by perusing the product specifications. I said a few. Most of the current microprocessor designs are too complex in control signal definition, timing, and instruction operation codes to be easily understood. The EA9002 has been oriented for the systems, control and logic engineer. The EA9002 architecture contains all those good things familiar to computer architects like ALU, registers, stack, RAM and program counter but organized to be readily understood by the non-computer type engineer. Similarly, the instruction set contains features like arithmetics, logicals, conditional and unconditional jumps, control flags and input/output commands. But the instructions are defined and organized in such a way that even a novice to the world of microprocessors can understand them.

A full complement of logic instructions as well as four rotate instructions makes for good bit handling capability. Conditional jumps can be made within a page (256 words) boundary. Unconditional jumps can be made anywhere. Six instruction operation codes provide the equivalent of 12 instructions by means of an arithmetic flag which automatically indicates the zero or non-zero state of the accumulator. This allows definition of instruction such as:

Jump if Accumulator = Register
Jump if Accumulator ≠ Register
Jump if Accumulator < Register
Jump if Accumulator > Register
Jump if Accumulator ≤ Register
Jump if Accumulator ≥ Register

Decimal arithmetic is performed without cumbersome software correction methods. All ADD and SUBTRACT instructions as well as INCREMENT and DECREMENT instructions can be executed in either binary or decimal arithmetic. In decimal mode, the eight bit quantities are treated as two decades of BCD digits. The resulting value in the accumulator is a corrected BCD pair. The execution of one or more additional BCD correction instructions is not required.

TTL compatibility - clean instruction execution timing - straight forward interfacing signals. All have been carefully planned and executed for EASE of comprehension but completeness in capability.

Has the job been well done. Initial market reaction indicates it has. It remains for time to provide a more complete answer. The EA9002 is by no means an optimum product. There is no such thing. Nor is the EA9002 the end product. It is just the beginning of a new breed of emerging products attempting to service an exploding market demanding new and better microprocessor products.

6-24-76

MICROPROCESSOR LANGUAGE DESIGN

Dr. Geoffrey C. Leach
Sycor, Inc
100 Phoenix Drive
Ann Arbor, MI 48104

Abstract

The programming language is a critical tool for making the microprocessor accessible for teaching, research or system implementation. Considerations of time and space, which are raised by the limited capabilities of presently available processors, make the systems implementation language of great interest to the microprocessor user. This paper presents some considerations in the design of such languages and examines the application of these principles to four existing languages.

Introduction

For whatever purpose a microprocessor is to be used - teaching, research or the implementation of commercial systems - the matter of access is crucial. The use of microprocessors today compresses twenty-five year's history of large systems; users of binary programming techniques coexist with users of Fortran and Cobol and all intermediate levels of language. In the implementation of systems for large and medium scale computers, the systems implementation language (SIL) has lately been recognized as a tool of considerable importance.[1] This paper presents some of the aspects of systems implementation languages and examines four manifestations: PL/M, PLuS, MPL and SYCLOPS.

The motivation for the viewpoint taken by this paper is simple. The author believes that the design and implementation of a microprocessor language is a worthwhile and productive project and one that will repay the required investment in increased future productivity. Furthermore, the resources required for such a project are not large and are generally available at most universities. All that is required is interest and direction in the right place.

Such a project has two obvious goals. Improved access to a particular microprocessor and educational experience in language design and compiler constructions. One might argue that the former goal is of such preeminent importance as to make the latter inconsiderable. The writer questions this. Not only is the design of a compiler a good way to become intimate with a machine, but the increased programming productivity that results makes the target system available to a much larger number of users. Finally, an opportunity is provided to use a language that helps cultivate good programming practices. This is something which is not available with any of the languages provided by the semiconductor manufacturers.

The design of SILs is not *per se* an area of active research. Rather, it is a by-product of research in programming language design. Consequently, this paper will raise more questions than it answers. It appears, however, that in the microprocessor user community there is a strong motivation to develop a theory of SILs that gives direction in the design of useful languages. It is well known that the software costs of large systems exceed the hardware costs, but only by a factor of 1.5 to 3. A microcomputer costing five thousand dollars clearly cannot afford software that costs a hundred thousand dollars to develop. The use of SILs is one way to attack this problem.

Thoughts on SIL Design and Implementation

A *Systems Implementation Language* (SIL) is one which provides facilities for the convenient expression of a restricted class of problems and which may be translated with reasonable effectiveness for a particular target machine.

By this definition, the first SIL to gain prominence was XPL[2], which was designed for writing compilers (the problem) on the IBM System/360 (the target).

Fortran II is a SIL for writing numerical applications on the IBM 704. A SIL is not a machine-oriented language (MOL). The MOL designer has as his goal the algorithmic structuring of the language while retaining complete control over the generated code. MOLs of interest to the workshop are ML80[3] and SM/PL[4].

Given the definition, how does one go about designing a SIL? Texts on language and compiler design are becoming quite numerous and the basic principles are fairly well established. The resources for building a SIL compiler are available in any computer science program, once students have completed the compiler construction course. A method of construction is needed, ALGOL, PASCAL and PL/I support recursive descent techniques. XPL and XPL/S[5] have the MSP[2] and SLR[7] techniques of table-driven syntax analysis available. There is also the LALR technique[8] for those with access to a /370 with PL/I. The resources are available.

It will be observed that the only SILs in this list are XPL and XPL/S. Compiler-writers are evidently no more eager to avail themselves of this technique than those they service. The other languages mentioned provide the facilities required for compiler construction. They also provide much that is not required. Fortran is absent from this list, despite the fact that most commercially available microprocessor language compilers are written in Fortran. The reason for the use of Fortran is portability of the resultant compiler. Fortran imposes upon the compiler writer a host of difficult problems not related to compiler construction and the result is much more difficult to maintain. Consequently, the writer urges that Fortran be considered only as a last resort. A good macro assembly language is probably a better choice.

The question of the portability of programs is important and must not be dismissed lightly. The language of implementation is not the only way to achieve this. The PL/M compiler offered by Intel, for instance, uses concepts and techniques developed by McKeenan and his coworkers for XPL. In fact, a knowledge of this work is important in the understanding of PL/M and its compiler as is a knowledge of Fortran.

One must decide what systems are to be implemented. Traffic light controllers? Compilers? Time-shared operating systems? Micro networks? Each class of problems has its own particular needs, which will determine the general scope of the language. Remember, the complexity of a compiler grows faster than the size of the language. The preferences of the designer or (perhaps) the users will determine the syntactic

form of the language. Numerous models are available: Algol 60/68, PL/I, Pascal, PL/M; even Fortran or Cobol! The guide to design must be to facilitate the statement of solutions to the problems of interest.

The choice of the details of the language, especially control and data structures, is a matter for negotiation between the problem space and the target machine. While the designer should try to avoid leading the programmer into large amounts of unanticipated computation. If one must multiply, a system subroutine to do the job is preferable to forcing the user to write his own: 'A * B' is easier to write than 'MULT (A,B)' and has more documentary content. How far one can go in this direction must be left to the designer. Ultimately, we would like to be able to write 'SOLVE'; practically we must operate at a somewhat lower level.

Perhaps the problem space is not so well defined. The question then becomes, "What set of general-purpose tools facilitate the solution of a reasonably broad range of solutions without introducing gross inefficiency?" Some answers to this question can be found in the examination of existing MPU languages.

Case Studies in Microprocessor SIL Design

The languages examined in the following sections, PL/M, PLuS, MPL and SYCLOPS, all attempt to provide a SIL for a particular MPU. The last three were all influenced by PL/M, whose preeminent role forces the questions of compatibility and transferibility upon even the most unwilling language designer. As a consequence we will present PL/M in some detail, to be followed by shorter comparative presentations of the others.

PL/M

PL/M (Programming Language/Micro)[9] was originally developed as a SIL for the Intel 8008[10]. The influence of XPL is quite clear. The instructions added to make the 8008 into the 8080[11] were such as to improve the efficiency of generated code, particularly in the manipulation of 16-bit quantities (generally addresses) but did not fundamentally change the machine, from the point of view of the SIL designer, so, we will not distinguish between the two. In what follows, we stress those features of the 8008/8080 architecture that give rise to corresponding language features.

Program Structure. The PL/M program is compiled as a single unit. There is no notion of independent compilation, or external procedures or data. Consequently, the entire program must be recompiled to effect the smallest alteration. Reference to absolute locations (data or code) although possible, is unattractive both from a program organization standpoint and from a teaching standpoint. Well-structured programs[12] are not supposed to use absolute addresses! What one gains from the single unit structure is freedom from an additional linkage editing step.

PL/M is a block-structured language. Identifier scope is coincident with procedure nesting and, within this, the nesting of simple do ... end groups. The latter is probably an unnecessary frill, provided that the storage so declared is not recovered at the end of the group. While the patterns of identifier use may be made clearer this way, the creation of a large number of small contexts detracts from clarity.

Lexical Properties. Identifiers up to 31 characters in length are permitted; sufficient for all but the most explicit! Character set restrictions have led to the use of the $-sign as a break character. This was an unfortunate choice - it is hardly inconspicuous and does not serve to separate components of a compound identifier well. USE$SAVE is not as clear as USE_SAVE. Although conspicuous to the eye, the $-sign is unnoticed by the compiler - USE$SAVE, USESAVE$, etc. are all equivalent. This contributes both confusion and a source of error that is difficult to detect.

The representation of numbers provides a choice of four radicies: Decimal, Binary, Octal and Hexadecimal. The specification technique used is to append a character, D,B,O,Q or H, to the digits. D is optional and Q is an alternative for O.

Comments are bracketed with /* and */, after XPL and PL/I. 360/370 users will agree that this continued use of a logical end of file (if '/' is in column 1) is unfortunate.

Control Structures. PL/M supports a good variety of control structures, most of which are built around the group concept.

A group is a sequence of statements bracketed by do and end. If there is nothing appended to the group head (do) a simple group results. The addition of while gives a do-while group, of case, a do-case group, etc. These compound groups are control structures and provide for iteration or selection.

The do-while group (do while A=B; ...; end;) permits control based on the evaluation of an expression (to true/false) on each iteration of a loop. If the expression is simple, this feature makes good use of conditional branch instructions.

The do-case group permits a conceptually efficient 1-of-n selection that is quite useful in practice. The 8080 does not support indirect addressing so the code generated for the case selection is less efficient then it might otherwise be.

The iterative-do group (do I=1 to 10 by 2; ...; end) provides an equivalent to the familiar Fortran do statement, with a test on entry. The computation of limit (to) and increment (by) is repeated for every iteration (when nonconstant). This may make ROM-based systems easier to build (although I doubt it) but it certainly provides an unexpected (to those familiar with XPL) overhead.

The if-then-else construction (if A=B then...else...;) is as useful as it is familiar. There are two alternative strategies for evaluating compound logical conditions in an if clause (if A & B & C then...). The one used by most languages requires complete evaluation, except where no side effects are possible. Thus, if A is false in the above example, extra code is executed. In addition, a logical result must be generated and carried along, which requires considerable additional code. A better strategy is to have the first false condition terminate the evaluation. This allows the load-and-convert-condition-code sequence to be replaced by a simple branch-on-condition-false.

The procedure call and return are the key to the structuring of PL/M programs. Procedure definition only provides for call by value, although a form of call by reference is available through the use of based variables. The availability of a stack on the

8080 makes its use for the passing of parameters expected. However, the 8080's inability to directly reference variables on or through the stack makes it less attractive. The interrupt procedure permits the programming of appropriately located interrupt service routines.

The goto statement provided is completely unrestricted. A goto may reference a label outside the procedure in which it appears, a variable previously assigned a target address or an absolute address. This provides the educator with an interesting quandary. Admittedly the goto is convenient, but it seems clear that it does not encourage good programming practice[13]. If one uses PL/M, one cannot avoid the goto. One can only discourage its use and attempt to teach principles of structured usage[14].

The ability to reference absolute addresses implies the ability to position programs in memory. PL/M provides this by means of a numeric label. (100: X = X+1; places the first byte of code at location 100).

The halt, enable and disable statements provide direct access to the corresponding instructions. A necessary feature, as no assembly language interface is provided.

Data Structures and Operations. The 8008 is intended for 8-bit logical operations; hence, the byte data type. An interpretation of the most significant bit as sign is provided by the MPU but not implemented in the form of a signed data type. 16-bit address quantities are possible, hence the address type. Additional 16-bit operations in the 8080 serve to strengthen this.

The H and L registers of the 8080 can be used for 0-origin subscripting, so a singly dimensioned array is provided. On the other hand, multiplication must be implemented in software, so additional dimensions, which might tempt the programmer into inefficiency, are not supported.

The H and L registers are also effective in the implementation of base register addressing, which gives rise to the based variable. No restrictions are placed on assignments to the basing variable, so it is simple to wander out of the allocated data area. There is no dynamic storage allocation associated with based variables.

The use of ROM in MPU based systems suggests the need for a read-only data type. This is provided by the data declaration, which provides named storage that is initialized at program load time. Other variables can be initialized as well, but the possibility of change means that they must be placed in RAM.

Data operations are conventional, for the most part. A full set of logical (and, or, xor, not), relational (=, > =, < =, >, <, < >) and arithmetic (+,-,*,/,mod) are provided. The '< >' operator translated to "not equal" and arises from the exclusion of '?' from the 64 character ASCII set. In addition, plus (add with carry) and minus (subtract with borrow) allow multiple byte arithmetic to be programmed.

Certain operations are implemented with built-in functions. The shift left/right (SHL/R), rotate left/right without carry (ROL/R) and with carry (SCL/R), all provide direct access to the corresponding MPU instructions, while concealing the fact that the instructions do not have a shift count and extra code is usually required. This is also the case for input-output. Access to the appropriate ports is provided by the INPUT and OUTPUT pseudo-variables. The argument to both is a constant port number. The absence of any facility for a variable latch (either in software or hardware) can create difficulty in program design and generate extra code.

The condition code bits are directly accessible through the CARRY, SIGN, ZERO and PARITY pseudo-variables. Care must be taken to insure that the code generated does not destroy the sense of their use.

Decimal arithmetic (and access to the 8080's decimal adjust instruction) is provided through the DEC built-in function.

As a compliment to the access of absolute locations with gotos, all of memory can be addressed as a single, contiguous space. MEMORY(I) runs from the first byte of memory to the last.

The 8080 stack pointer (SP) register can be directly addressed via the STACKPTR pseudo-variable.

Conclusion. PL/M is a blend of constructs influenced (or required) by the 8080/8008 architecture and constructs influenced by programming practice. It is clearly an experimental language - a first cut, perhaps. It is unfortunate that its popularity is fast enshrining these features in a state of arrested development. We do not yet know enough about MPU programming to be able to tolerate a standard language, but this role is often proposed for PL/M.

PLuS

Programming Language Micro, Signetics[15] may be regarded as a reconsideration of PL/M by its original author, Gary Kildall. It is not a "second generation" language, for it is constrained by a necessity for compatibility with PL/M. As such, it is interesting to consider PLuS in constrast to PL/M.

The Signetics 2650[16] microprocessor is intended for the same applications as the 8080/8008. That is, applications where the 8-bit data item predominates. The architecture set is highly regular and features indexed, relative and indirect addressing as well as register, immediate and direct with a full compliment of logical, arthimetic and conditional control instructions and autoincrement indexing. The address space is 15 bits (32K) and there are no double-byte operations.

Program Organization. PLuS permits the independent compilation of procedures which are linked by a system loader. Inter-procedure communication (also with assembler subroutines) is implemented with global and external declarations, which apply to both procedures and data. The addition of a common facility would provide the compiler with the opportunity to improve code in the relative addressing mode.

The absence of double byte operators requires a considerable number of software supported operations. The required calls are effectively implemented with the two byte 0-relative subroutine call instruction.

Lexical Features. These are identical to PL/M.

Control Structures. One structure is added to the PL/M set, the end-until group. The simple group do... end, becomes do...end until (expression) and is executed until the expression yields a false value. This provides control at the end of the group, which is useful in situations requiring one or more executions. Do-while and end-until cannot be combined.

As with the do-while, end-until makes good use of conditional branching instructions if the expression is simple.

The indirect addressing feature permits efficient implementation of the case statement.

Data Types and Operations. To the PL/M data types, fixed(7) and fixed(15) have been added. These represent 8 and 16-bit two's compliment signed numbers. Variable and data initialization has been extended to include address expressions, so it is possible, for instance, to build a transfer vector at load time.

Some architectural restrictions are apparent. Indexed access is limited to data on the same page. Otherwise, a procedure reference is required. This will undoubtedly lead to some very strange programs, which are highly sensitive to small modifications. However, it is in the spirit of avoiding excessive code generation.

Conclusion. A reasonably successful attempt has been made to improve on PL/M while retaining compatibility.

MPL

The Motorola Programming Language definition[17] was available to the author only in preliminary form and may have changed substantially. Should this be the case, it would be an interesting exercise to compare our criticisms with the changes.

MPL is targeted at the M6800 microprocessor[18]. In addition to the usual register, immediate and absolute addressing modes, it provides based, relative and direct (to the first page) modes.

MPL produces assembly language output. This offers a possibility to hand-optimize a program. It also offers the possibility of never returning to the original source and, in effect, reverting to assembly language.

Program Structure. No mention is made of separate compilation of global/external data. Escape to assembly language is possible and with the intermediate assembly language provides the necessary tools to simulate separate compilation.

Lexical Properties. Identifiers are restricted to six characters and there are no break characters. These limitations are derived from the use of the assembler.

Numbers can be represented in hexadecimal and binary notation, as well as decimal. Paired double quotes (") may be used for hex strings.

A logical end of line character (!) has been added to the PL/I comment brackets (/* and */).

Control Structures. In consideration of the amount of attention being devoted to the improvement of control structures, it is acutely distressing to note regression in this area. MPL replaces the case statement with Fortran's (PL/I's) computed goto.

Procedure calls are implemented with parameter call by reference (which may not be passed on to other procedures) rather than the usually less costly call by value. A special form of call is supported, when the parameters fit neatly into registers. It is unfortunate that such a "kludge" needs to be embedded in the syntax; the compiler, after all, has the required information. The do-while and iterative-do groups are provided in much the same way as PL/M. They may be combined, after PL/I, to yield a statement of the form: do I = 1 to 10 while A B.

Along with the computed goto mentioned above, an assigned goto is provided. This provides the same facility as a PL/M goto which references an assigned address variable.

Data Structures and Operations. In contrast to its limited control structures, MPL provides a good variety of data structures.

Logical types, bit(1) and bit(8), signed binary(1) (7 bit) and binary(2) (15 bit) and optionally signed decimal to 12 digits provide a full range of types that should cover most applications. A character type is also supported, but operations are limited to assignment and comparison.

Two varieties of structure are provided: arrays of up to three dimensions, using 1-origin subscripting, and hierarchical structures (PL/I type) of up to five (contiguous) levels. The data description facility is somewhat limited by the restrictions placed on subscripting: subscripts must be binary(1) variables, constants or simple expressions (including variable ± constant). These restrictions are clearly derived from the architecture, but are obvious in the light of the need for subscript calculation with no multiplication. Based variables are available with a pointer operator, which permits local designation of a base. 'P → A' is interpreted as "P points at (is the base of) A. The usual operations are provided.

One feature that bears mention is the address "operation". In PL/M, for instance, the address of A is written .A. In MPL it is written 'A'. How then, does one distinguish between the address of A and the string containing A?

No mention is made of built-in function or pseudovariables, including any for input-output. Presumably, this is due to the closeness of assembly language and the belief that one-instruction items might as well be generated in line.

Conclusion. One might say that MPL is an attempt to put a recognizable PL/I on an MPU. This would certainly explain the choice of some language features.

On the other hand, if a SIL was intended, then the success of the effort must be judged on the efficiency of the generated code and the ease of programming.

SYCLOPS

SYCor's Language for OPerating Systems[19] has been developed at Sycor, Inc. To facilitate software development for the 440 Clustered Terminal System,[20] which is based on the Intel 8080. To date, a considerable volume of software, including control programs, access methods and even a Cobol compiler, has been written in a Sycor-modified version of PL/M. We believe, however, that PL/M is unattractive as a long-term programming tool.

There were two major reasons for our decision to replace PL/M. Control of our development language and removal of 8080 specificity. In the first instance, we want to be in a position to improve and experiment. The language described here is only a first step. In the second, we feel that we must be in a position to take advantage of new MPU developments as they arise, not when the semiconductor manufacturer gets around to supporting PL/M.

As implied by the name, the designs that are implemented in SYCLOPS are systems-level programs. These are characterized by a frequency of execution such that size and speed are of concern and a level of complexity that makes assembly-level programming unattractive from a productivity standpoint. (We are, by the way, doing some assembly language work, but only to recode already debugged code that we have identified as critical). Our long-term objectives are the same as the short-term. Increased programmer productivity and improved code through improved language and compiler design, while retaining sufficient machine independence to permit the transfer of systems-level software.

SYCLOPS is implemented in BLISS-10.[21]

Program Structure. A SYCLOPS compilation is a series of units of two types: Procedures and shared data blocks. The former differ from the normal procedure in that they are implicitly global in scope (they may be called from other, separately compiled procedures) and may have the attribute main, which defines program start address.

A shared data unit is similar to a Fortran labeled common block. However, once encountered, shared units remain known to the compiler until cleared by a control command. This enables several procedures to be compiled independently with access to the same data. This is important because the memory addressing architecture of many MPUs allows code optimization, if two bytes are known to be on the same page. The declaration of global and/or external data is also possible, but here the relative positions are not known. We have removed the group-scope facility provided by PL/M, as it was never used.

Lexical Properties. These are basically the same as PL/M. SYCLOPS permits up to 32 characters in an identifier (one has to stop somewhere) and allows $, %, ¢, #, _, ? and @ as (significant) break characters. The representation of numbers is also similar, except we do not permit O for octal. To the comment brackets, we have added the ! as a record terminator. In this we follow the advice of Plauger[22], but the practice is also followed in Bliss and MPL.

Control Structures. The SYCLOPS control structures are similar in purpose to those of PL/M and PLuS. However, they have been altered in form to stress more clearly the structuring of the program by using matched statement terminators. Unlike Bliss and Algol 68[23], we do not spell things backwards. The structures available are:

if ... then ...[else]... endif

step ... endstep

do ... enddo [unless]...

while ... endwhile [unless]...

case ... endcase [otherwise]...

procedure ... endprocedure

Labels are optional for all but the procedure, where one is required. When a label appears, however, it is required on the matching end. All structures may have empty bodies.

There is no question that the structures are more verbose than those of PL/M and that the requirement of a matching label will cause "unnecessary" errors. We believe that they will improve the readability and maintainability of the code that will more than compensate. One difficult-to-find error of mismatched ends will more than compensate for a few easy-to-correct label errors.

The goto statement has been retained for compatibility. We are committed to its eventual banishment from the language and have already eliminated absolute labels and goto targets. To facilitate the removal of the goto, we have added a repertoire of structured escapes. These provide the ability to leave a compound statement (exit) repeat from the beginning (repeat) and go to the next iteration of a step statement (nextstep).

The PL/M data types have been revised and extended. The address data type has been replaced by word, which we define as the largest directly addressable item of primary storage. For the 8080 it is 16 bits. These types may, optionally, be signed. Thus a signed byte is 7 bits plus sign, equivalent to the MPL's binary(1) and the PLuS fixed(7).

The singly dimensioned array remains the only structure offered. The size of the array can be specified as *, which indicates that exactly the storage required by the initialization of the declaration is reserved.

SYCLOPS declarations may not be factored. It is our belief that this will encourage better documentation practices.

Hierarchical structures have not been implemented due to the pressure of time. Such structure as can be accessed efficiently will be provided.

Data operations include the standard set, with the obvious semantic extensions to cope with signed variables.

Conclusion. We have begun our move away from the PL/M syntax with changes in control structures. We will follow this with changes in data structures. What is equally significant for the purposes of this workshop, is that we are doing this with relatively small resources (a 3-person team, about two man years in all). Additionally, "compatibility" is not a strong consideration. Given that we are prepared to translate from SYCLOPS to the 8080 machine language, it is not difficult to use the same program components to translate from PL/M to SYCLOPS.

One aspect of the SYCLOPS project that has not been mentioned is that of code generation. We expect to achieve a 30% improvement over present PL/M compiler, merely by improving the design of the code generators. Following this, a source level optimization phase will be interposed. The structure of the project was designed with this in mind; consequently it will be a straightforward task. Contrast this with the task of modifying a vendor-supplied compiler coded in Fortran.

Summary and Conclusions

We have examined some of the principles of SIL design and seen their application in four languages designed for and used in the production of MPU software. None are completely adequate.

It has been suggested that PL/M should be adopted as an industry standard for Microprocessors. Indeed, a survey of available languages[24] indicates a strong trend in the direction of making PL/M a proper subset, as was seen in PLuS. It is hoped that this paper has provided some inuication that this is a premature

conclusion. Not only is the problem space ill-defined (every day new MPU applications appear) but the architecture of MPUs is advancing rapidly. (Compare the 8008 and the TMS 9900[25]).

It will soon be the case that it is practical to design special purpose microprocessor architectures for particular problems. These special MPUs will require even more specialized languages. To insist otherwise would be to sacrifice effectiveness on the altar of uniformity.

References

1. Proceedings of a Sigplan Symposium on Languages for Systems Implementation. Sigplan Notices 6,9 (October, 1971).

2. McKeeman, W.M., J.J. Horning and D.B. Wartman, A Compiler Generator, Prentice-Hall (1970).

3. Pedroso, L.R.B. ML:80: A Structured Machine-Oriented Microcomputer Programming Language, Naval Postgraduate School, December (1975) NTIS AD-A020 055/0WC.

4. SM/PL is a language for the National Semiconductor Corporation: PACE microprocessor. It has not yet been formally released.

5. Leach, G.C. XPL/S: A Systems Implementation Language for the Xerox Sigma Computers , University of Washington, Department of Computer Science Technical Report 74-02 (February 1974).

7. DeRemer, F.L. "Simple LR(K) Grammars", Comm A.C.M. 14,7 (July, 1971).

8. Lalonde, W.R. "An Efficient LALR Parser Generator", University of Toronto, Computer Systems Research Group Technical Report CSRG-2 (April 1971).

9. 8008 and 8080 PL/M Programming Manual, Intel Corporation (1975).

10. "Intel 8008 Microcomputer Systems User's Manual" Intel Corporation (1975).

11. "Intel 8080 Microcomputer Systems User's Manual" Intel Corporation (1975).

12. Yourdon, E. Techniques of Program Structure and Design, Prentice-Hall (1975).

13. "The Goto Controversy" in B.M. Leavenworte, ed. Special Issue on Control Structures in Programming Languages, Sigplan Notices 7,11 (November, 1972)

14. Knuth, D.E. "Structured Programming with goto Statements", Computing Surveys 6,4 (December, 1974).

15. "PLuS Reference Manual", Signetics Corp. (March, 1976).

16. "Signetics 2650 Microprocessor", Signetics Corp. (1975).

17. "MPL Language Reference Manual", Motorola Microsystems (1976).

18. "M6800 Microprocessor", Motorola Microsystems (1974).

19. "Sycor Language for Operating Systems", Sycor, Inc. (1976).

20. "Sycor 440 Clustered Terminal Processing System", Sycor, Inc. (August 1975).

21. "BLISS-10 Programmer's Reference Manual", Digital Equipment Corp. (1975).

22. Plauger, P.J. "Signal and Noise in Programming Languages" , Proceedings ACM 1975 Conference.

23. Van Wijngaarden, A (Ed.) "Report on the Algorithmic Language ALGOL 68" Springer-Verlag (1969).

24. Leach, G.C. "The State of the Art in Microprocessor Software", Proceedings Electro 76 (1976).

25. "TMS 9900 Microprocessor", Texas Instruments, (October 1975).

THE AUTOMATED GENERATION OF CROSS-SYSTEM SOFTWARE FOR MICROCOMPUTERS

Gearold R. Johnson
Associate Professor
Mechanical Engineering Department
Colorado State University
Fort Collins, Colorado

and

Robert A. Mueller
Scientific Programmer
Texas Instruments, Inc.
Dallas, Texas

Abstract

GEN is a software system comprised of a set of independent FORTRAN program writer modules designed to generate microcomputer assemblers and simulators. It is simple enough to be used by those with limited architecture and programming backgrounds but flexible and powerful enough to generate efficient and well-structured assemblers and simulators for microcomputers with sophisticated architectures and instruction sets. This paper presents a brief description of the generating system, the generated assemblers and simulators and the advantages they offer in both programming and pedagogical applications.

Introduction

When looking at the historical development of computer systems, three distinct observations are apparent: 1) Computer systems are physically getting smaller and smaller. 2) The cost of computer systems is decreasing continuously and nearly exponentially. 3) The frequency of new computer systems introduced on the market is definitely increasing. The present paper addresses the second and third of the above points in terms of software; that is, with systems becoming less expensive all the time, how can software costs be kept in line with hardware costs; and, secondly, with the increased frequency of new products on the market, how is the user to be able to write and develop software for these new machines as they become available? Both of these points were seriously aggravated by the introduction of microprocessors in 1971.

The third point of the first paragraph led to the development of the software to be discussed. The increased frequency of new computer systems, particularly microcomputer systems for use in dedicated systems or as embedded processors, has forced the design engineer to become thoroughly familiar with the machine assembly language programming and hardware characteristics. Then, it was necessary for him either to obtain access to hardware phototyping systems such as are marketed by Intel, Motorola, National, and many others or to gain access to high level language based assembler/simulators that each of these manufacturers has written for their representative devices and are available either by purchase from the vendor or over nationwide timesharing networks.

Either approach can cost thousands of dollars! The development of microcomputer or minicomputer simulators using students or programmers for the program development is another alternative (1). In fact, one of the authors spent two months writing an Intel 8008 assembler and a simulator (2) for use on a CDC 6400 in order to develop software packages for a data acquisition system (3). The author was rewarded for his endeavor upon completion of the package by the introduction of the Intel 8080 chip and then, of course, the subsequent introduction of approximately twenty other microprocessor chips which are currently available. This led the authors to contemplate the development of a software system that would generate assemblers and simulators (4) for each microprocessor.

The overall philosophy of the software generation system is that of a register transfer language but at the function or instruction level rather than at the gate level such as the hardware description languages like CDL (5), IDDAP (6), etc. The software generator also enables the user to concentrate on the definition of his target system without worrying about internal programming problems. The user may generate an assembler/simulator for any particular device which may or may not exist and, having once generated this program, is free from its actual operation and can move on to the more pressing problem -- software development for the particular machine. This automated generator presents no problem in terms of inefficiency because of the infrequent use of such code as compared with, for example, FORTRAN compilers or other large scale operating systems because the time saved generating the assembler/simulator far outweighs the inefficiencies of the actual generating system. It allows the user to save tremendous amounts of time in building assemblers and simulators for various machines and puts him immediately into the task of software and hardware development. To develop a system simulator and assembler using the automated software generator is a one to five-man-day effort rather than what has traditionally been a two to six-man-month effort.

The objective was to construct a software package that would enable the user to automatically generate an assembler and a simulator for any given small computer system very quickly in order that he get on to the main task at hand -- software development utilizing that system. It was also felt imperative that any such system developed be machine independent and, therefore, the GEN system was written in standard FORTRAN IV. This approach would be far outweighed by the time saved in actually getting a device simulated and to proceed with program development.

Design Considerations of the Simulator Generating System and Generating Process

Some of the major objectives in the design and development of SIM/GEN were to make both the generator and the generated programs as simple to use and simple to modify as possible without losing a great deal of generality. To achieve these goals, a highly modular approach was taken in the generator system design.

The operation of all modern general-purpose digital computers is based on the repetitive sequence of FETCHing the next instruction. The initial goal of generating a simulator was divided into two subtasks: that of generating a fetch and decode (DECODE) module and an execution (EXECUTE) module. The DECODE module is well-defined and small enough to not have to be broken down further. EXECUTE, however, spans a wide

variety of tasks and appeared to be far too extensive to be handled by a single unit.

The technique to treat the EXECUTE phase was to group the instruction set into classes in which all the instructions in a particular class matched identically in their component bit structures. That is, all instructions with an opcode of length i, operand field one of length j, operand field two of length k, etc., and which all have m operand fields would be grouped in a single class. The DECODE routine could fetch the next instruction's operation code, determine in which class the instruction belonged, branch to that module, and then the decoding of the operand parts could be done at the start of the module without regard to which instruction in the module was being executed. This seemed to be an effective solution and required that the user define only one module at a time, totally independent of the remainder of the system.

With the program architecture laid down, the next problem was how to generate a well-structured program that allowed the user flexibility in the definition of the microprocessor without having any concern about the simulator program to be produced. It was decided that both the generator and generated simulator(s) would be implemented in FORTRAN IV. This offered a natural way to keep generator phases independent by making each generated module a FORTRAN SUBROUTINE which can be compiled independently. The run-time intermodule transfer of information is accomplished via COMMON blocks which allow the addition of simulated I/O devices by simply using the right COMMON interfacing between SIM/GEN and the externals.

The actual program to be generated was written as a "skeleton routine," that is, the program was complete except for the constants defining the target machine's architectural dimensions and particulars that are "filled-in" from the user input. Within the skeleton are "markers" indicating that user input is required to complete a missing part at the marked point. The generating process then becomes a matter of first copying the skeleton to some disk file and then having the generator transfer the skeleton to another file, card by card, filling in the missing parts with user-provided data at the appropriate places. The latter file would contain the complete module upon termination. Diagnostics are included to aid in correction of syntax errors, however, cases of misformatted values can result in a complete module with incorrect machine specifications. For this reason, a detailed manual was written to provide explicit instructions and diagrams describing each module and its input data layout. (7)

Note that the process of "feeding" the SIM/GEN generators a few constants and appropriate system-provided skeleton routines appears to be quite simple. This is a feasible means of handling the DECODE module but a more descriptive and powerful means of specifying the instruction set is compulsory for the system to remain effective. It was decided that a definition language consisting of micro-level operators and architecture component operands was the answer with the user essentially going through the steps of "microprogramming" each instruction. (Husson (8) defines microprogramming as "a systematic approach and an orderly approach to designing the control section of any computer system.") Thus, IDL (Instruction Definition Language) was developed. It consists of 29 primitive operators resembling commonly used assembly language mnemonics, 16 basic operand designators, 6 relational operators for conditional statements, provisions for specifying constants to any radix r, $2 \leq r \leq 16$, and the definition of subroutines to eliminate redundant coding of frequently needed sequences of IDL statements. Tables I through IV present the basic IDL information.

To define an instruction, the user provides a verbal description which is translated into a Comment card, the operation code value, the execution time for the instruction, and a sequence of IDL statements which perform the desired target machine operation. An example is shown in Figure 1. These are the RAL and RAR instructions for the Intel 4040.

```
VERBAL EXPLANATION:  RAL  ROTATE LEFT
      OPCODE VALUE:  F5
    ESTIMATED TIME:  10.8
    IDL STATEMENTS:
         SHLC SREG1  SREG1  1
         AND TEMP4 SREG1 1
         AND SREG1 SREG1 $16
         OR SREG1 SREG1 SREG2
         MOVE SREG2 TEMP4
         ENDINSTR

VERBAL EXPLANATION:  RAR  ROTATE RIGHT
      OPCODE VALUE:  F6
    ESTIMATED TIME:  10.8
    IDL STATEMENTS:
         AND TEMP4 SREG1 1
         SHRL SREG1 SREG1 1
         SHLL SREG2 SREG2 3
         OR SREG1 SREG1 SREG2
         MOVE SREG2 TEMP4
         ENDINSTR
```

FIGURE 1

Many machine instructions can be emulated with a single IDL statement while others, like a Decimal Add, can be rather complicated. Almost any conceivable instruction can be emulated using the available operators and operands provided, but some may require some ingenuity on the part of the user.

IDL offers no control structures other than the simple conditional IF. This precludes the use of "looping." It was our belief that inclusion of fancy control structures would only cloud the issue and would have little application in the instruction definition process. As a consequence, all generated code is executed in a straight-line fashion giving the simulators a highly desirable structure.

The final design consideration was concerned with the memory configuration and communication with the CPU (EXECUTE modules). The typical use of LSI memories with microprocessors leads to microcomputers with small blocks of memory (RAM or ROM) at discontinuous address locations on the address bus. This is quite different from the traditional single monolithic blocks of memories used in most large-scale and minicomputers. To efficiently facilitate the simulation of such a structure, the SIM/GEN system used an internal virtual memory mapping scheme in order to store many discontinuous blocks of memory (of varying type) as a single internal contiguous block.

The memory description is given in the third SIM/GEN module, the MEMORY module. The user may define his particular configuration by stating his memory blocks as non-overlapping ordered pairs of the form

$$(\ell_i, h_i)$$

or

$$(\ell_i, h_i, R)$$

where ℓ_i is the low-end address and h_i the high-end address of the i^{th} memory block and R signifies that the memory block is Read-Only. The access of memory during simulator execution is through the Data and Address Busses found in the IDL operand repertoire. In addition to the memory configuration, register counts and stack descriptions are also given as the MEMORY module input.

An architectural view of SIM/GEN 5.3 and the generating process is given in Figure 2.

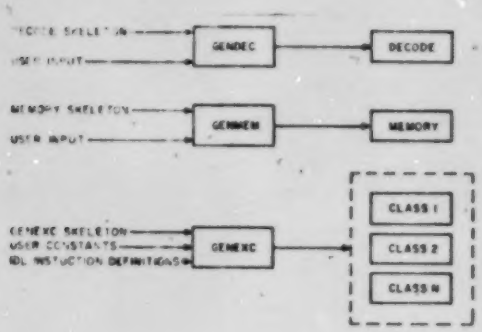


FIGURE 2. SIM/GEN Generating Process

The Generated Simulators

As previously mentioned, the generated simulators are written in FORTRAN IV. Each module is implemented as a SUBROUTINE with all inter-module information transferred via COMMON blocks. This allows the user to simulate I/O devices and interface them by using the proper COMMON blocks in the I/O routines. An operational block diagram of the simulator is shown in Figure 3.

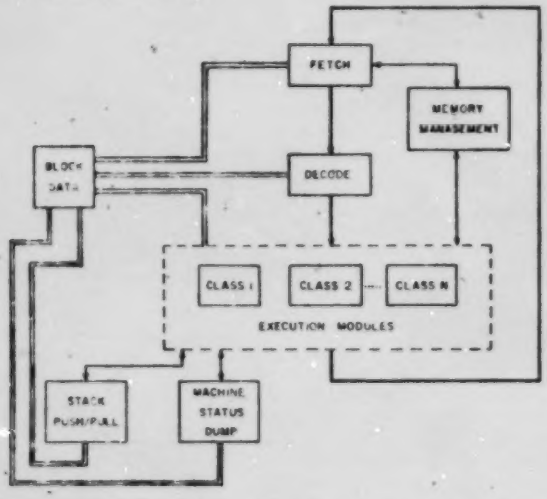


FIGURE 3. Simulator Operational Block Diagram

We will now examine the internal operation of each of the modules and show how certain machine characteristics can be handled naturally.

To start, assume that an instruction has just finished execution and it is now time to fetch the next instruction. The operation of the FETCH routine of the DECODE module is shown in Figure 4. It performs the function of fetching the next instruction's operation code. Since many microprocessors are multiword instructions, this may involve fetching a sequence of memory words. It is the user's job during the generation process to specify how many memory words need to be fetched to accomplish this phase. The instruction is transferred from memory to the Instruction Register (IR) to be used later in the decoding and the appropriate EXECUTION module.

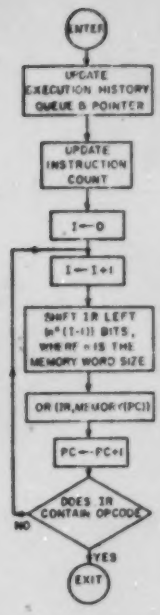


FIGURE 4. Flowchart of FETCH Routine

Also, the contents of the program Counter are placed in front of a circular queue with eight entries and the queue pointer is adjusted. This provides an execution history of the last eight instructions executed which can be displayed by invoking the Machine Status Dump Module to be discussed later. This option offers the user a valuable diagnostic aid in determining the cause of a program failure since the machine status is printed upon normal termination as well as premature termination due to an error.

The second phase is performed in the decode routine of the DECODE module shown in Figure 5. Its purpose is to extract the operation code, to determine whether an illegal operation is being attempted and, if not, to direct the simulator to the appropriate EXECUTION module where execution of the desired operation is carried out.

Control is now transferred to one of the EXECUTION modules. Figure 6 presents the flowchart of an EXECUTION module. Remember that all instructions in a given EXECUTION module match identically in their component bit structures as described in the previous section. Therefore, the first step in the EXECUTE module is to divide up the remainder of the instruction and place the values of each of the operand fields in contiguous elements of an operand array. After the extraction process OPR (1) would contain the contents of the first operand field, OPR (2) the contents of the second operand field, . . ., and OPR (n) the contents of the n^{th} operand field, where all instructions in the EXECUTE module under consideration have n operand fields.

This is very convenient since IDL then allows the user an IMMi operand designator where IMMi will be generated into OPR (i). The algorithm used for the extraction process is shown in Figure 7. Note that it requires no special provisions for variable-length instructions or the case in which operand fields extend across word boundaries.

With all operands decoded, the next step is to branch to the section of code which executes the desired operation. This is done using tables as in the decoding to obtain a branch table index for control

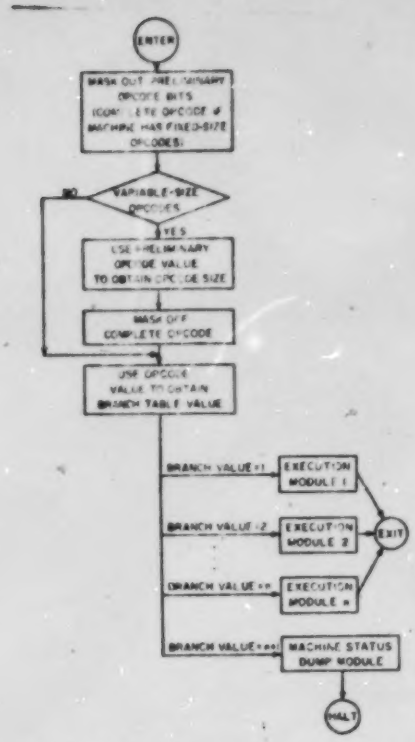


FIGURE 5. Flowchart of DECODE Routine

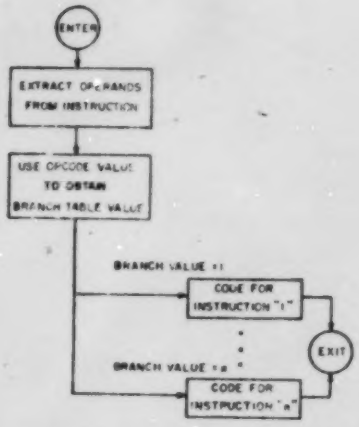


FIGURE 6. Flowchart of EXECUTION Module

transfer to the appropriate code section for the instruction with operation code. The final steps are the execution of the code which simulates the instruction's operation, the exit from the module, and restarting the complete sequence in the fetch phase.

Perhaps the most important feature a computer system simulation offers is its ability to display an informative run-time synopsis of the machine status at discrete steps during program execution. The information displayed by the SIM/GEN MACHINE STATUS display module includes the contents of the Program Counter, ROM [Program Counter], the Instruction Register, the Data Bus, the Address Bus, the Elapsed Execution Time, the number of executed instructions, the contents of the Stack Pointer, Stack, the memory contents, and the execution history queue.

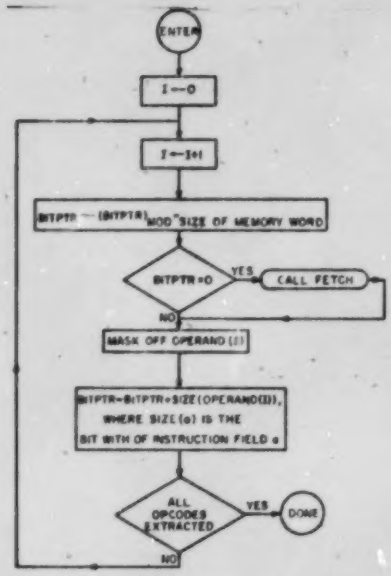


FIGURE 7. Flowchart of OPERAND Extraction

Assemblers and Cross-Assemblers

An assembly language is a symbolic form for describing a computer program for a specific machine. That is, it provides a set of mnemonic names for describing machine instructions, allows the use of names for representing addresses, keeps track of addresses, and provides convenient ways to minimize time and effort spent in the development and debugging of a computer program. Thus, it offers a more powerful means of expressing and comprehending a program without any loss of contact available at the machine language level.

An assembler is a translator which takes as input the symbolic assembly language program and produces as output the machine language (binary) equivalent form which can be directly executed by the computer. Another common translator is the compiler which is associated with languages which are problem or algorithm oriented whereas the assembly language is a machine oriented language.

Note that an assembler is written to run on the machine for which it generates the specific machine language code, i.e., it is machine dependent. The notion of cross-assembler refers to an assembler that is written to run on a computer other than the computer for which it generates machine language code. This implies generally that the code generated cannot be executed on the machine where the assembler is resident.

A natural question might be "what good is the machine language code if it cannot be executed?" or "why not just run the assembler on the computer for which the machine language code is generated?"

There are several answers which motivate the uses of the cross-assembler, particularly when the target machine language is for a microprocessor. First of all, there is the problem of storage requirements. A well-facilitated assembler might require a considerable amount of memory in order to run as a resident program. Many small computers are lacking this storage space. Secondly, one would probably like to implement the assembler in a higher-level language. Higher-level languages that run on minicomputers are becoming more common; however, they usually lack many of the powerful options offered by compilers resident on larger

machines. The use of microprocessor resident compilers for higher-level language programming is virtually non-existent at this point in time. A third consideration is the availability of input-output devices for communicating with the small computer. The common modes of I/O for such machines are slow teletypes with paper tape readers and punches as opposed to high-speed card readers and line printers and the mass-storage back-up devices such as magnetic tape drives, disk drives, and drums which are a part of all large scale computer systems. Perhaps the most important use of the cross-assembler is for supporting the programming on the computer simulation which is presumably resident on a large scale system. With the cross-assembler and test computer simulator, one can completely develop and test his software with all the conveniences offered by the larger computer system. This can be a big time saver and can lower the cost of software development drastically.

Given that the use of such cross-system software development aids are desirable and that microprocessor simulations can be generated with relative ease, we also need a means of generating assemblers with which to program the generated simulators. The ASM/GEN Assembler Generating System (9) was developed for this purpose.

ASM/GEN and the Meta-Assembler

The ASM/GEN system is based on the same principles as the SIM/GEN system in that it is given, as input, the specifics for a given machine's assembly instructions and produces an assembler which will translate the symbolic assembly language program to its machine language equivalent.

The generated assemblers offer the well-known macro generation facilities, conditional assembly, user-defined symbols, radix changing, text string specification, and the use of arithmetic expressions. The system uses two-pass, absolute assembly, and allows the user a somewhat free format. The decision to use absolute addressing was based on the fact that very few microprocessors have resident operating systems and that most serve a dedicated purpose (that being the continuous execution of a single program).

The drawing in Figure 8 illustrates the basic operation of the ASM/GEN system. It takes as input a description of the assembly source language and object translation and produces as output an assembler for that machine. Note that all psuedo-operations (non-executable instructions) are invariant throughout all generated assemblers. They vary only in their assembly instruction mnemonics and the binary (object) translations. Thus, the source language and object translations need be specified only once to ASM/GEN and the resulting output is a cross-assembler which can be used in conjunction with a SIM/GEN simulator (or any other simulator) to develop and debug microprocessor programs.

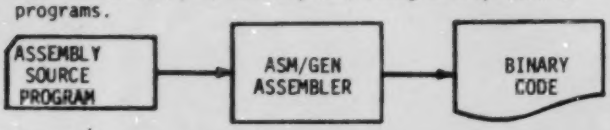


FIGURE 8. The Assembly Process

The ASM/GEN Generating Process

In order for the ASM/GEN to be capable of producing a complete assembler, one must provide some descriptive information. It must be informed of the machine word size so as to be able to keep addressing straight and know the bounds for constants. The type of arithmetic used is assumed to be two's complement and constants are represented in this form. A list of all instruction mnemonics must be given with the binary translation value. This information is stored in the symbol table. Lastly, the instructions are classified in a scheme similar to that used under the SIM/GEN system. That is, all instructions which have an identical component bit structure are grouped within an assembly class. The structure of each class is described using simple ordered fields which state field bit position and width.

The inputs are then used to complete the ASM/GEN skeleton and the resulting output is a complete assembler. A pictorial view of the assembler generating process is given in Figure 9.

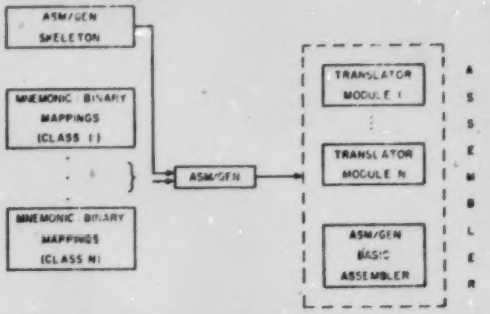


FIGURE 9. ASM/GEN Generating Process

Applications

The GEN system of FORTRAN program writer modules has been used extensively. Assemblers and simulators have been generated for the following microprocessors:

INTEL 4004	MOTOROLA M6800
INTEL 4040	NATIONAL PACE
INTEL 8008	MOS 650X
INTEL 8080	FAIRCHILD F8

SCIENTIFIC MICROSYSTEMS MICRO-CONTROLLER (SIGNETICS 8X300)

The time required to generate the assembler and simulator varied between one day for the SMS microcontroller and a week for the National PACE.

It would appear that the automated generation of microcomputer software has at least two uses in the university environment. First, it provides a tool for instructing students in elementary computer architecture and/or microprocessor systems. The simulator generating process requires that the user have a good understanding of the basic working of general-purpose digital computers and have a thorough register-level familiarity with the target machine to be simulated. Having students describe simulators for carefully selected machines could be very edifying. Since the generation of a simulator can be accomplished in from one to two weeks after the student has been initially introduced to the machine, it is reasonable to assume that this is a feasible approach. Tables V and VI show possible course outlines for computer architecture and microprocessor systems.

The other academic area of application is in research endeavors. The simulation of computer systems is a widely accepted means of doing implementation, testing, and data gathering in a multitude of applications. The ability to generate a working set of cross-system software which includes a microcomputer assembler and simulator in less than a week's time (for the more experienced researcher) is highly desirable. It allows the researcher to expend the majority of his funds and energy towards accomplishing a given task of which the need for microcomputer simulators is consequential. Work is being conducted at

Colorado State University which is aimed at increasing microprocessor software efficiency through improved hardware organization from the standpoint of "ease of programming." The ability to accurately simulate and modify a variety of hypothetical and actual microprocessors with little time and effort requirements has proven to be invaluable.

Many other research projects have benefited through the use of the National PACE microprocessor as an on-line velocity profile display system for blood flow in arteries using a Doppler ultrasound velocity measuring device. Another project uses an MOS 6502 microprocessor in a data acquisition system for acquiring and analyzing data in a solar heated and cooled building (10). The INTEL 4004 has been used as the controller for an astrographic drive on a telescope (11). An INTEL 8080 has been used as the controller for three reservoir basin models for water resources control (12). Each of these projects has used GEN generated systems successfully to develop the resident software required.

Conclusions

Both GEN generators and the generated assemblers and simulators are written in ANSI FORTRAN IV (with the exception of few small character manipulation routines), thus providing a highly portable package. The use of an instruction definition language offers the user a set of "micro-level" operators and architectural component operands with which he essentially microprograms the instruction set. Since most vendor literature provides this type of descriptive information about the instruction set, the definition process becomes a relatively simple one.

The initial desire to be able to have a working assembler and simulator for a microprocessor within a few days after obtaining the preliminary vendor literature has now been realized. The degree of effectiveness of GEN as a pedagogical tool can only be ascertained with time.

References

(1) Fisher, D. D., "Simulators and Assemblers for Minicomputers," Proceedings of the Symposium on the Simulation of Computer Systems, III, Boulder, Colorado, August 12-14, 1975.

(2) Johnson, G. R., "An Assembler and Simulator for the Intel 8008," an unpublished user's manual, Colorado State University, Fort Collins, Colo., July, 1973.

(3) Johnson, G. R., J. C. Fugitt, and D. L. Starner, "A Data Acquisition System Based Upon the Intel 8008," an unpublished paper, Colorado State University, Fort Collins, Colorado, 1975.

(4) Mueller, R. A. and Johnson, G. R., "A Generator for Microprocessor Assemblers and Simulators," Proceedings of the IEEE, Vol. 64, No. 6, June, 1976.

(5) Chu, Y., "Introducing the Computer Design Language," Proceedings of the Sixth Annual IEEE Computer Society International Conference, San Francisco, September, 1972.

(6) Crall, R. F., "IDDAP -- Interactive Computer Assistance for Creative Digital Design," Ph.D. dissertation, Electrical Engineering Department, University of Missouri, Rolla, 1970.

(7) Mueller, R. A. and G. R. Johnson, SIM/GEN 5.3 User's Manual, University Computer Center Report 76-1., Colorado State University, Fort Collins, Colorado, November, 1975.

(8) Husson, S. S., Microprogramming: Principles and Practices, Prentice-Hall, 1970.

(9) Mueller, R. A. and G. R. Johnson, ASM/GEN 5.3 User's Manual, University Computer Center Report 76-2, Colorado State University, Fort Collins, Colorado, January, 1976.

(10) Johnson, G. R. and C. B. Winn, "Smart Thermostats for Minimizing Energy Consumption," Proceedings of the Eighth Annual Simulation Symposium, Tampa, Florida, March, 1975.

(11) Bourque, J. G., "An Astrographic Telescope Drive," unpublished paper, Colorado State University, Fort Collins, Colorado, 1975.

(12) Keilian, G., C. B. Winn, and G. R. Johnson, "Closed Loop Controller for a Multi Reservoir Problem," an unpublished report, Colorado State University, Fort Collins, Colorado, 1975.

TABLE I

IDL STATEMENT SYNTAX AND SEMANTIC INTERPRETATION

IDL STATEMENT	SEMANTIC INTERPRETATION
ADD oprl opr2 opr3	oprl ← opr2 + opr3
AND oprl opr2 opr3	opr ← opr2 · opr3
CALL oprl opr2 ... oprN.	CALL oprl (opr2,...,oprN) N≤8
CLEAR oprl	oprl ← 0
COMONE oprl opr2	oprl ← $\overline{opr2}$
COMTWO oprl opr2	oprl ← $\overline{opr2}$ + 1
CONCAT oprl opr2 (n) opr3	oprl ← opr2 (bit-width of opr2=n) concat opr3
DECR oprl opr2	oprl ← opr2 - 1
DEFINE oprl opr2 ... oprN.	SUBROUTINE oprl (opr2,...,oprN) N≤8
DISPLAY 'text'	WRITE "text" on printer
DUMP	CALL STATUS dump
ENDCLASS	End of Execution class input
ENDIRSDEF	End of Execution class instruction definitions
ENDIRSTR	End of machine instruction or subroutine definition
EXIT	Branch out of machine instruction
HALT	STOP
IF oprl rel-op opr3	IF [oprl <rel-op> opr2] then execute following IDL statements until ENDIF
INCR oprl opr2	oprl ← opr2 + 1
MOVE oprl opr2	oprl ← opr2
OR oprl opr2 opr3	oprl ← opr2 v opr3
PULL oprl	STACKP ← STACKP + 1
	oprl ← STACK(at STACKP)
PUSH oprl	STACK(at STACKP) ← oprl
	STACKP ← STACKP - 1
RETURN	RETURN from subroutine
READD	Data Bus ← Memory Word (at Address in Address Bus)
READI	Data Bus ← Memory Word (at Address in Memory Word at Address in Address Bus)
SET oprl	oprl ← $2^{\text{word size}} - 1$
SHLC oprl opr2 opr3	oprl ← opr2, rotated left opr3 bits
SHLL oprl opr2 opr3	oprl ← opr2, shifted left opr3 bits (zero fill)
SHRA oprl opr2 opr3	oprl ← opr2, shifted right opr3 bits (sign extended)
SHRL oprl opr2 opr3	oprl ← opr2, shifted right opr3 bits (zero fill)
WRITED	Memory Word (at Address in Address Bus) ← Data Bus
WRITEI	Memory Word (at Address in Memory Word at Address in Address Bus) ← Data Bus
XOR oprl opr2 opr3	oprl ← (opr2 v opr3) · (opr2 · opr3)

TABLE II

IDL RELATIONAL OPERATOR MNEMONICS

MNEMONIC	EXPLANATION
EQ	EQUAL TO
GE	GREATER THAN OR EQUAL TO
GT	GREATER THAN
LE	LESS THAN OR EQUAL TO
LT	LESS THAN
NE	NOT EQUAL TO

TABLE III

IDL OPERAND MNEMONICS

MNEMONIC	EXPLANATION
ABUS	ADDRESS BUS
DBUS	DATA BUS
[REG.j]	GENERAL REGISTER #j
GREG.i	GENERAL REGISTER # [Immediate Operand Field i]
IMM.i	IMMEDIATE OPERAND FIELD i
PC	PROGRAM COUNTER
SREG.k	SPECIAL REGISTER #k
STACK	STACK, Address = Stack Pointer
STACKP	STACK POINTER
TEMP.i	TEMPORARY REGISTER i
integer	DECIMAL INTEGER CONSTANT
integer . r	INTEGER CONSTANT TO BASE "r"

$1 \leq i \leq 8$

$0 \leq j \leq$ Number of General Registers - 1

$0 \leq k \leq$ Number of Special Registers - 1

$2 \leq r \leq 16$

[n]: This notation signifies the "contents" of n.

: Number

TABLE IV

IDL OPERATION MNEMONICS

MNEMONIC	EXPLANATION
ADD	ARITHMETIC ADD
AND	LOGICAL AND
CALL	CALL SUBROUTINE
CLEAR	CLEAR TO ZERO
COMONE	ONES (LOGICAL) COMPLEMENT
COMTWO	TWOS COMPLEMENT
CONCAT	CONCATENATE
DECR	DECREMENT
DEFINE	DEFINE SUBROUTINE
DISPLAY	DISPLAY LITERAL TEXT
DUMP	MACHINE STATUS DUMP
ENDCLASS	END OF EXECUTION CLASS INPUT
ENDIF	END OF IF BLOCK (THEN-PART)
ENDINSDEF	END OF INSTRUCTION DEFINITIONS
ENDINSTR	END OF MACHINE INSTRUCTION OR SUBROUTINE DEFINITION
EXIT	BRANCH TO END OF MACHINE INSTRUCTION
HALT	MACHINE STATUS DUMP AND STOP
IF	CONDITIONAL IF
INCR	INCREMENT
MOVE	DATA TRANSFER
OR	LOGICAL INCLUSIVE OR
PULL	PULL OFF STACK
PUSH	PUSH ONTO STACK
READD	READ MEMORY DIRECT
READI	READ MEMORY INDIRECT
RETURN	RETURN FROM SUBROUTINE
SET	SET TO ALL (LOGICAL) ONES
SHLC	SHIFT LEFT CIRCULAR
SHLL	SHIFT LEFT LOGICAL (ZERO FILL)
SHRA	SHIFT RIGHT ARITHMETIC (SIGN EXTENDED)
SHRL	SHIFT RIGHT LOGICAL (ZERO FILL)
WRITED	WRITE MEMORY DIRECT
WRITEI	WRITE MEMORY INDIRECT
XOR	LOGICAL EXCLUSIVE OR

TABLE V

CS 350 COMPUTER ARCHITECTURE COURSE OUTLINE

I. General Organization of a Computer
   - ALU
   - Registers
   - Memory
   - Busses
   - Control
   - Input/Output

II. Introduction to SIM/GEN
   - Specification of a computer's architectural components
   - Memory and address specifications
   - Instruction Classification
   - Instruction Definition Language

III. Computer Control
   - Interpretive Execution Cycle
   - Definitions of Instructions
   - Microprogramming
   - Addressing Schemes

IV. Machine Arithmetic
   - Number Representations
   - (One's, Two's Complement, Signed Magnitude)
   - Fixed Point Multiplication and Division
   - Double-Precision Arithmetic
   - Floating Point Representations
   - Floating Point Arithmetic

V. Internal Character Representations and Conversions
   - BCD, ASCII, EBCDIC, CDC Display Code
   - Conversion Schemes

VI. Input/Output
   - Devices and Their Characteristics
   - Interfaces and Device Controllers
   - Polling Control of I/O
   - Interrupt Control of I/O
   - Buffering

VII. Representative Computer Systems
   - CDC 6000 Series
   - IBM 360/370 Series
   - Burroughs B5700/B6700 Computers
   - Super Computers
   - Mini Computers
   - Micro Computers

TABLE VI

CS/EE/ME 495NX MICROPROCESSORS APPLICATIONS COURSE OUTLINE

I. General Organization of a Microprocessor Element
   - ALU
   - Registers
   - STACK
   - Memory
   - Busses
   - Multiplexers
   - Control
   - Input/Output

II. Introduction to the GEN Cross-Products
   - Specification of the Microprocessor Architecture
   - Memory and Address Specifications
   - Instruction Classification
   - Instruction Definition Language

III. Representative Microprocessor Systems
   - INTEL 4000 and 8000 Series
   - Motorola M6800
   - National PACE and SC/MP
   - Signetics 2650
   - MOS 650X
   - Scientific Microsystems Microcontroller
   - Fairchild F8

IV. Microcomputer Structure
   - Interfacing the Microprocessor Element
   - ROM, RAM and I/O Organization
   - Clocks
   - Peripheral Interfacing Chips
     - Parallel Data Transmission
     - Serial Data Transmission
   - Interrupt Features and Polling Concepts

V. Bit-Slice Architecture
   - Concepts and Organization
   - INTEL 3000 Series
   - AMD 2900 Series

VI. Microprocessor Applications
   - Direct Digital Control
   - Digital Filters
   - Digital Data Acquisition Systems
   - Instrumentation in Bio-engineering
   - Device Controllers
   - Automotive Applications
   - Video-graphics

Interactive Interpreters for Microprocessor Systems

Yaohan Chu
Department of Computer Science
University of Maryland
College Park, MD 20742
301-454-4245

ABSTRACT This paper proposes the use of interactive interpreters for high-level language microprocessor systems instead of compilers. It introduces three types of interactive interpreters and describes briefly the highly interactive direct-execution interpreter. Current implementation of such a highly interactive system on a microprocessor system is also described.

1. Introduction

We now have two computer industries. The first computer industry is old. It is centered around major computer centers and major computer manufacturers. The center of gravity of this first computer is the "200 billion dollars" of software. Because of this enormous investment, the first computer industry has a strong resistance to changes and innovations, unless the hardware is built to make use of the existing software.

The second computer industry is new. It is represented by microprocessor manufacturers and the numerous small system companies. It is called the second computer industry, because the hardware manufacturers are different from those of the first computer industry. The markets are different. The applications are different. The system producers are different. The second computer industry is receptive to new ideas.

Although the second computer industry is different to the first in almost every respect as mentioned above, their approach to software follows that of the first computer industry, particularly in the use of compilers and assemblers. It has been widely recognized that software cost is becoming the bottleneck to the potential, large-scale use of microcomputer systems. Something new in software has to be conceived and developed to overcome this difficulty. Why not re-examine the balance and the trade-offs among hardware, software, and language instead of traditionally considering software alone? Why not abandon the approach of "one-system-and-one-language" for many applications that has prevailed in the first computer industry? Why not develop simple high-level languages for particular applications instead of "general-purpose languages"? Why not build microprocessors with an architecture for interpreters instead of for compilers? Why not use interpreters instead of compilers?

2. Compilers or Interpreters

Since the arrival of the Fortran Compiler during the middle of the 1950's the compiler has made a great contribution to the ease of computer programming and thus to the large increase in computer applications. The use of the compiler has become so popular that it has been dominating the computer world for the past 20 years.

However, the compiler creates the need for other software such as linkage editors and loaders. Since many languages (high-level language, assembly language, relocatable code, and core dump) are involved in a compilation process, a competent Fortran programmer really needs to know all of these languages to some extent. This requirement of knowing these languages causes additional difficulty in debugging. Therefore, Fortran programming is not as ideal as was first thought when it was introduced.

Since their arrival, compilers have become greatly complex partly because more complex language constructs have been created. The computer manufacturers are becoming weary of implementing any new and powerful high-level languages because of the cost and time to develop and maintain the compilers. Yet, programming language is a central issue; more programming languages are bound to come. It appears that compilers and high-level languages are now on a "collision course".

Historically, interpreters came just before the arrival of compilers. Interpreters have not become popular because the computer systems of 20 years ago were too slow. Since then, there has been a great advance in computer technology. One of the most significant contributions in hardware is the enormous increase in computer speed. The justification for ignoring the interpreter does not exist any more. It is time to look again at the idea of interpreters. After all, computation cannot be achieved by a compiler since a compiler only translates from one language to another. Computation must be accomplished by an interpreter (either in software or hardware).

For the past 20 years, more time and effort have been spent on the research and development of compilers than on interpreters. Thus, it is fair to say that more knowledge exists about compilers than interpreters. However, it is known that interpreters are easier to write, partly because they need no code generation, and generation of efficient code is a most difficult task in compiler writing. Furthermore, most of today's high-level languages are compiler-oriented languages; they are not the most suitable for efficient interpretation. Since the users of an interpreter need know no language other than the high-level language itself, the interpreter is much easier to use and is more amenable to an interactive system.

3. Three Types of Interactive Systems

An interactive high-level-language system is one where the user can interact with the system by means of a high-level-language on a terminal. This terminal allows the user to enter lines of characters and to edit these lines for corrections and changes. The terminal can interrupt the system and thus receive a quick response. Such a system is superior for programming, particularly program debugging because program errors could be found quickly.

There are three kinds of interactive high-level-language systems: interactive compilation systems, interactive interpretation systems, and interactive direct-execution systems. An interactive compilation system is an interactive high-level-language system in which a compiler is employed. It allows the source code to be entered at the terminal, and a text editor is available for making changes and corrections as the source code is being typed at the terminal. After the entire source code has been input, the compiler is called to translate the source code into a relocatable code. During the compilation, the syntax of the source code is checked and the syntax error messages are later printed out. If the compilation is successful, the linkage editor is next called to link the user's relocatable code with the subroutine relocatable codes and to output an absolute code. During linking, external references are checked and error messages are output. Finally, the absolute code is loaded into the memory and is then executed. This process of compilation, linking, and execution is repeated until the program is completely debugged. Note that the user's interaction is limited to the level of the entire source code.

An interactive interpretation system is an interactive high-level-language system in which an interpreter is employed. The high-level-language is the programming language, and the user needs no assembly language. There is neither compilation, nor assembly. The user writes only high-level language software. Such a system allows the source code to be input on the terminal and a text editor is available for making changes and corrections as the source code is being typed on the terminal. After the entire source code has been entered, the interpreter is called to syntax check and interpret the source code. Instead of the entire source code, the interaction can alternatively be carried out at one-line a time. As a result, this system provides more interaction between the user and the system. The interactive APL system is an example of this type.

An interactive direct-execution system is an interactive high-level language system in which an executer (or a direct-execution interpreter) is employed. As each symbol of the source code is being entered at the terminal, the symbol is syntax-checked and executed. This is accomplished by the "interactive direct-execution loop". An error message is printed out as soon as an error is found. This process of symbol-by-symbol entering, checking, and execution gives a high interaction between the user and the system. Such a system also assists the user in debugging logical errors by showing the partial result of execution whenever it is requested. It also allows the user to command the system to execute the accumulated source code from the beginning, displaying the result at specified places. When the source code is completely entered, it would have already run once and could have been largely debugged.

4. Interactive Direct-Execution System

A direct-execution system executes a high-level-language program directly. It does not translate the source program into an object code before its

execution. An interactive system allows the user to use the terminal to communicate directly with the system. Thus, an interactive direct-execution system allows immediate checking and then correcting of lexical and syntactical errors as they occur without producing the "error snowballing" as happens in a conventional compilation run. The direct execution can also assist the user in debugging logical errors. As the source program is being entered on the terminal, the errors are being checked and corrected. When the program is completely entered, the source program could be largely debugged.

(a) System Language

An interactive direct-execution language may consist of four parts: a monitor language, a text editing language, an input processor language, and a high-level-language; each is interpreted by an interpreter. For the pedagogical interactive system to be described, these are rather simple languages.

(b) System Configuration

The system configuration is shown in Fig. 1. There are eight system units: the Monitor, Input Processor, Text File, and Teletype. The Monitor controls the operation of all system units directly or indirectly. All of the inputs from and the outputs to the Teletype are handled by the I/O Processor. The Scanner is called by the Direct-Execution Interpreter only.

The Monitor operates in four modes: Monitor mode, Input mode, Edit mode, and Run mode. After the system is initialized, the system is in the MONITOR mode and the symbol "@" appears on the terminal. There are four monitor commands: INPUT, EDIT, RUN, and SCRATCH. The user normally begins his programming by entering the INPUT mode by using the INSERT command. It returns to the MONITOR mode from any other mode by entering the special escape character on the terminal.

When the INPUT processor is called, it responds by printing "INPUT" at the terminal. It is then ready to accept a symbol from the terminal. When a symbol arrives, the Input Processor accepts it followed by the End-of-Text character and places them in the Text File where the source program is stored. It then calls the Direct-Execution Interpreter for syntax checking and interpretation. If the token is in error, the interpreter prints an appropriate error message on the terminal and so indicates by setting switch ERRTOG. If the token is the second occurence of the character "$", this indicates the end of the source program; the interpreter so indicates by setting switch PDONE. In any event, the Micro-Algol interpreter returns to the INPUT processor.

The INPUT processor now tests whether switch PDONE is set. If it is, the INPUT processor returns to the MONITOR. Next, it tests whether switch ERRTOG is set. If it is, then the INPUT processor resets indicator ERRTOG and requests another token from the user to replace the one in error. Thus, the user only retypes the token that was in error and does it immediately after the error is committed.

If the user wishes to rerun with new data a program that is already in the source text buffer, the user enters the RUN mode and then the Micro-Algol interpreter re-executes the program.

When an error is found, the user can enter the EDIT mode to modify the source program. This program can then be rerun by entering the RUN mode. If the entire program must be scrapped and rewritten, the user can use the Monitor ERASE Command to do so and then enter the INTERACTIVE mode to write another program.

In short, as the source program is being entered on the terminal, its syntax is checked and the errors are corrected. When the program is completely entered, the source program could be debugged.

(c) System Control

The interactive direct-execution system described here has a monitor which controls the selection and activation of the micro-Algol interpreter, the text-editing interpreter, and the input processor as well as printing system error messages. The control cycle of this monitor is shown in Fig. 2. Letters E, I, R, and S represent the previously mentioned monitor commands. Letter X is the exit command of the text editor, and special character "ESC" represents the escape command of the Input processor. There is an inner loop in Fig. 2. This loop indicates where the interactive direct-execution occurs. This is the loop where the interaction occurs between the Micro-Algol interpreter and the input processor.

5. Implementation on a Microprocessor System

The advent of commercial microprocessors by Intel, Inc. in 1971 was a revolutionary event. Since then, many semiconductor device manufacturers have introduced microprocessors. Applications which were regarded as impractical in the past can now be implemented. However, the microprocessors of today are difficult to program. As a result, the microprocessor makers provide such software as cross assemblers,

cross compilers, and simulators. Even with the software support, the user still faces the need of a larger computer system for the use of the software. The impetus of this computer hardware revolution is dampened because of the cost of large software development. Is it possible to have a high-level-language microprocessor system without the need for a software maze? In the search for an interactive high-level-language system, the above described interactive direct-execution system can be implemented on a microprocessor system to realize an Interactive High-Level Language Direct-Execution Microprocessor System. Current implementation of such a system on an Intel Microprocessor 8080 system is now briefly described.

(a) Data Structure

The major data structures used by the Direct Execution Interpreter are shown in Fig. 3. The Text File is where the source program is stored as the tokens or symbols are being entered at the terminal by the user. There are two sets of pointers, one for the Input Processor and the other for the Interpreter.

There are stacks N and O. Stack N stores the operators during the processing of a statement. Stack O stores the operands during the evaluation of an arithmetic expression.

There are table D and stack BL. Table D is the symbol table, where the name and the value of each declared identifier in the Program File are stored. Since the language is block-structured, the entries of table D are allocated upon the entry of a block and deleted upon the exit of a block. The block-level stack BL keeps track of the block structure of the user's program.

There are stacks CS and WS. Stack CS is used by the Interpreter for the nesting of statements. Stack WS is used for the user's program in storing the return addresses of the PROCEDURE and WHILE statements.

(b) Control Structure

The control structure of the Direct-Execution Interpreter interprets the user's source program. It is shown in the flow chart of Fig. 4.

The source program begins with the two tokens in concatenation '$BEGIN' and ends with the two tokens in concatenation 'END$'. After initialization, the Interpreter checks out the two beginning tokens, calls the Error Routine if they are not correct, and returns to the Input Processor. If the program begins correctly, the Interpreter checks to see whether token 'BEGIN' is again encountered; this means the entry of a nested block or a nested compound statement. When 'END' is recognized by the compound tail routine, any local storage is released. The interpretation of the source program is completed when the two ending tokens are recognized; the Interpreter now returns to the Input Processor.

(c) Memory Allocation

The allocation of the 16K bytes of the Intel 8080 system is shown in Fig. 5. The memory is divided into two portions, the system areas (including the interpreter) and the user areas. The user areas consist of the text file, the data area, and the I/O area. The system areas consist of the system code area, the data area, the error message area, and the I/O area. The system code area takes about 8K bytes, because of the "built-in" debugging facility. The error messages take about 1K bytes and the text file 4K bytes for sixty-four 64-byte source text lines.

(d) Implementation Considerations

The microprocessor system is relatively inexpensive. However, inexpensiveness alone is not adequate for many applications where the system must be easy to use. An interactive high-level language direct-execution system can make a microprocessor system easy to use.

In order to realize an interactive high-level language system, the direct-execution type as described above offers a maximum of interaction between the user and the system. Some considerations for implementing such a system are discussed below.

(1) The system code size is an important consideration because the system code together with the system data areas and error messages must be in the memory all of the time (though the system code and error messages can be in the read-only memory).

(2) The speed of execution is another important consideration. Since the microprocessor system is interactive and for one user at one time, the speed may not be critical during program entry and program debugging. This is because as each token is being entered, the processor needs to execute only one syntactical procedure. However, if the execution of a loop is required, then a great deal of time would be required. This situation could be improved by eliminating syntax checking during execution of a loop once the program is debugged. On the other hand, the system is for one user at one time; there is no time-sharing overhead time.

(3) Another consideration is the debugging facility. There are two types. The first type is directly for the user's program; this includes syntax tracing and value tracing of variables. The second type is for those who understand how the Direct-Execution Interpreter works. This includes an indication of the current code and a dump of the Interpreter's pointers, stacks, and tables.

(4) Meaningful error messages are particularly important to the interactive system. No codes for errors should be used as they would defeat the purpose of the interactive system.

Most of today's microprocessor systems are implemented for the dedicated users where once implemented no further programming may be needed. However, for systems which require program changes or reprogramming, the use of an interactive high-level language microprocesssor systems is worthy of consideration.

6. Conclusion

There is little doubt that microprocessor systems will have the fastest growth ever experienced in the computer industry in the forthcoming years, particularly if the bottleneck of software could be removed. The use of the direct-execution interpreter combined with microprocessor technology could offer a solution to the current problems of high software cost, and the second computer industry could become even larger than the first one.

7. References

(1) Butler W. Lampson, "Interactive Machine Language Programming", Proceedings of Fall Joint Computer Conference, AFIPS Press, 1965, p.473-481.

(2) Boehm, B.W., "The High Cost of Software", Proceedings of a Symposium on the High Cost of Software, Monterey, California, Stanford Research Institute, September, 1973.

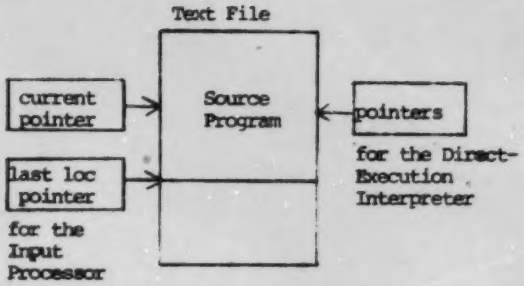
(3) Yaohan Chu, J.C. Yeh, and E.R. Cannon, "A Direct-Execution Interpreter", Technical Report TR-355, Department of Computer Science, Univ. of Maryland, February, 1975.

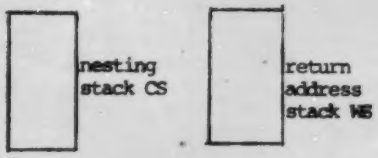
(4) Yaohan Chu and E.R. Cannon, "Design of an Interactive Direct-Execution System," Technical Report, TR-385, Department of Computer Science, University of Maryland, June 1975.

(5) Jim Gibbons, "When to Use High-Level-Languages in Microcomputer-Based Systems", ELECTRONICS, August 7, 1975, pp. 197-111.

(6) Chu, Yaohan and E.R. Cannon, "Interactive High-level Language Direct-Execution Microprocessor System", IEEE Transactions on Software Engineering, June 1976. pp. 126-134.

(7) Chu, Yaohan, (editor and author), High-Level Language Computer Architecture, Academic Press, Inc., New York, 1975.





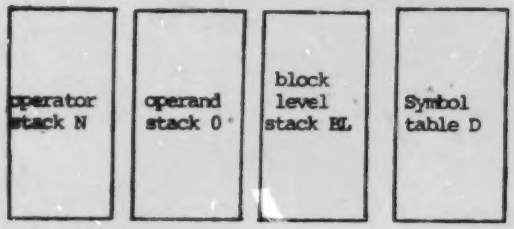


Fig. 3 The major Data Structures of the Direct-Execution Interpreter

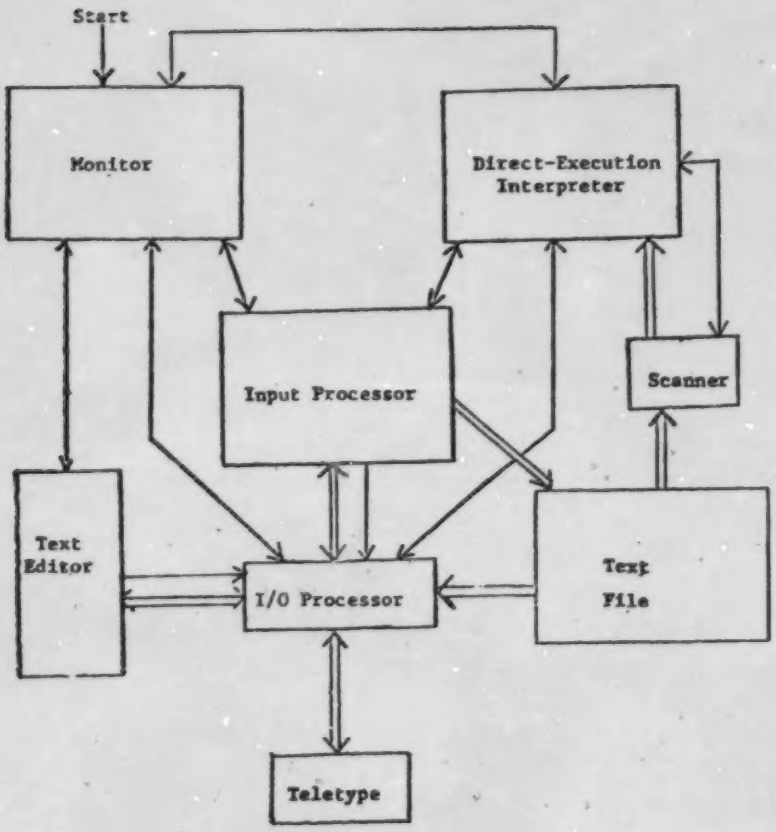


Fig. 1 Configuration of an Interactive Direct-execution High-level-language System
(double lines indicate data flow; single lines control flow)

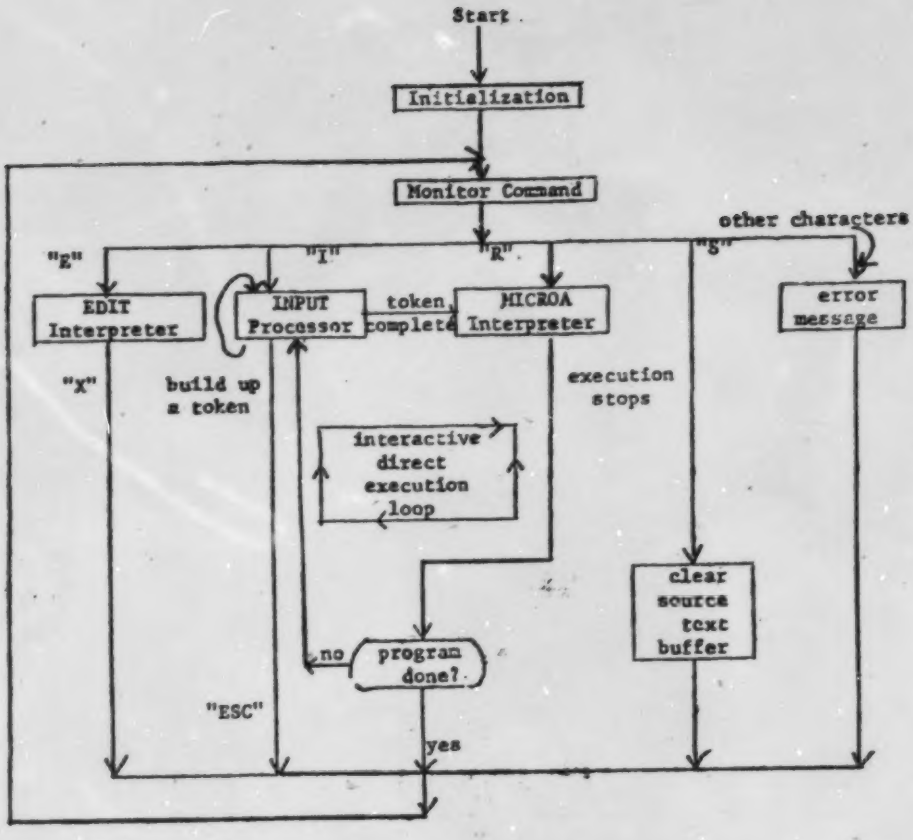


Fig. 2 The Interactive Direct-Execution Loop

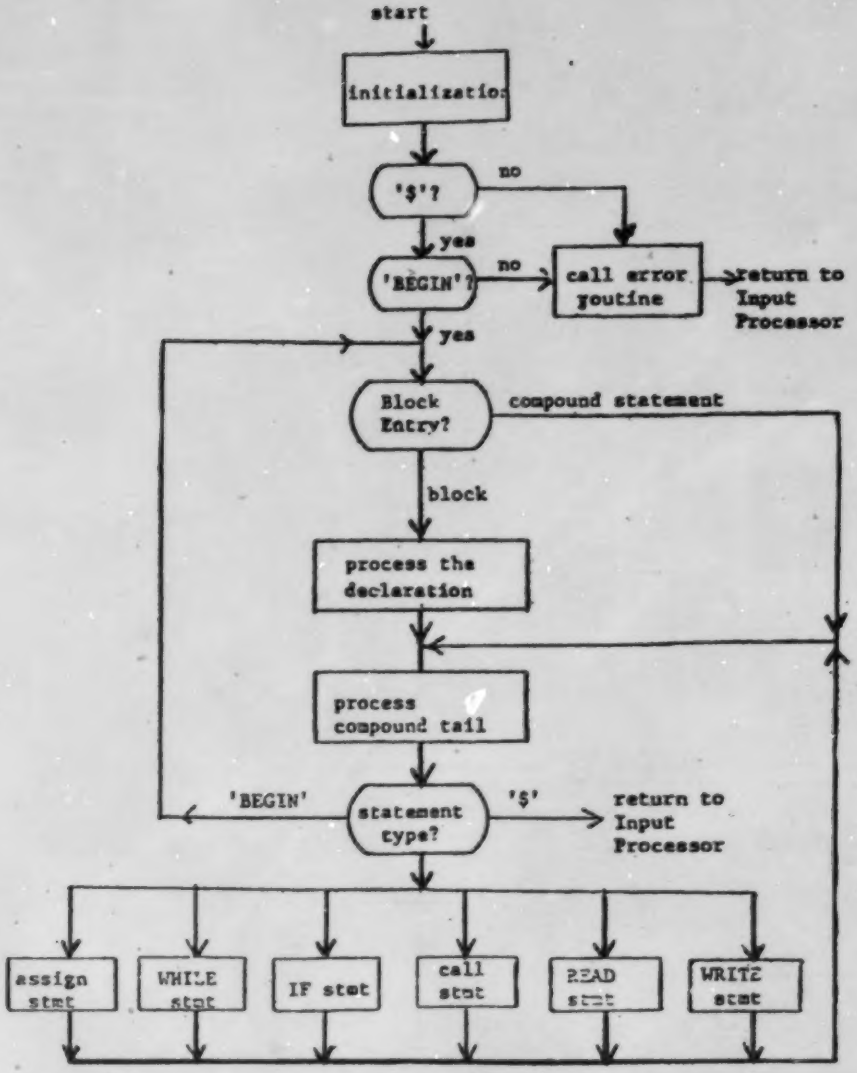


Fig. 4 Control Structure of the Direct-Execution Interpreter

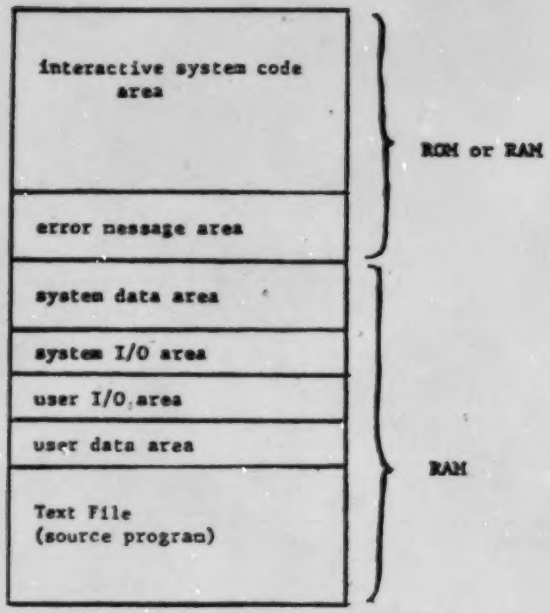


Fig. 5 Memory Allocation of the Intel 8080 System

INTRODUCTION OF MICROPROCESSOR EDUCATION INTO AN EXISTING CURRICULUM TO SATISFY UNIVERSITY AND INDUSTRIAL NEEDS

Dr. Ronald L. Krutz
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Abstract

As a building block, the microprocessor is portrayed as an alternative to random logic. In essence, it is much more than that. In its different forms, the microcomputer is a relatively sophisticated computational element, a programmable controller, a message handling and data switching device, or bulk storage administrator. True, random logic can be designed to perform these functions, but relatively, the microprocessor is a higher level building block than can be found in traditional random logic families.

Because of its many capabilities, the microprocessor will be applied by all engineering disciplines and by professionals in the sciences. In addition, it is finding increasing usage in industry.

Since the microprocessor educational needs of the computer engineer, mechanical engineer, physicist, and industrial employee are different and distinct, not one, but a number of educational approaches to teaching the microprocessor have been implemented at Carnegie-Mellon University. These different approaches along with the motivation behind them are detailed in the paper. Specific hardware and software used in laboratory sessions are discussed along with potential problem areas. Conclusions from courses already conducted are presented. Examples of microprocessor course outlines, a microprocessor problem assigned in a senior control course, and a student microprocessor project are given in the Appendices.

Introduction

The emergence of the microprocessor as an alternative to random logic in instrumentation, control, interfacing, and small scale data processing is a driving force for its introduction into the university curriculum. But, in contrast to many technological developments, the microprocessor, like the minicomputer, will find application in numerous professions. Because of the interdisciplinary nature of microprocessor applications and the different hardware and software skill levels required by the various disciplines to effectively utilize its capabilities, a spectrum of microprocessor educational approaches is being implemented at Carnegie-Mellon. These approaches recognize the following distinct groups of potential microprocessor users and/or designers:

- Undergraduate/Graduate Students
  * Electrical Engineers and Computer Scientists who will design and/or implement microcomputer systems at the very detailed hardware or software level and who require a formal in-depth sequence of computer/logic-design courses;
  * Electrical Engineers and Computer Scientists who work or will be working in areas not directly related to digital design such as control engineering or communications;
  * Engineers of other disciplines such as Mechanical, Chemical, Civil, Biomedical, etc., incorporating the microprocessor into their work, for example, process control and production automation;
  * Physical Scientists such as Physicists, Chemists, etc., who will be applying microcomputers to their experiments, instruments, equipment;
- Industrial Personnel
  * Industrial professionals whose organizations can benefit from the use of microprocessors in their products and/or operations.

Approaches

In order to meet the different needs of the above groups, microprocessor education has been introduced into the Carnegie Institute of Technology (CIT) curriculum by:

- modification of the content of existing digital course sequence;
- introduction of microprocessor topics into specific individual courses;
- development of new courses;
- student projects.

These four approaches were required since different emphases in the particular courses or projects were needed and the coherence and philosophical integrity of the existing digital course sequence had to be maintained. A brief description of the specific courses and projects involved follows:

- Modification of Content of Existing Course Sequence:
  * 18-233 Introduction to Digital Systems: covered Basic Digital Concepts, Circuits, Register Transfer Level Design, Introduction to Computers
  * 18-234 Computer Structures: Logic Design of Computers, Basic Architecture
  * 18-743 Computer Organization and Real Time Processing: Organization and Use of Minicomputer, Real Time Processing
  * 18-745 Minicomputer Systems: Minicomputer Project Course
  * 18-750 Advanced Computer Structures: Advanced Concepts in Computer Architecture
  * 18-859 Advanced Topics in Computer Engineering: Selected Advanced Topics

The need to introduce microprocessors more strongly into the above course sequence and yet retain the general approach to computer organization and software was recognized, and as a result, the following modified course sequence was adopted:

  * 18-233 Introduction to Digital Systems: Basic digital concepts, introduction to switching theory, introduction to RT level design, introduction to microprocessors, introduction to the minicomputer. The course would also include a set of self-contained, self-paced laboratory exercises from which each student would select two. Lab exercises in: (1) Switching Theory, using the DEC logic labs, (2) Integrated Circuit lab, (3) Computer-aided Design, using PDP-11 CAD package currently under development, (4) RTMs, (5) Microprocessors, (6) RTM minicomputer (C.rtm-L). The student would be required to work one lab from (1), (2), and (3) and another from (4), (5), and (6).
  * 18-234 Computer Structures: This course still

is concerned with the logic design of computers, but with more emphasis on logic design to compensate for the scaling down of logic design in 18-233, and less emphasis on computer architecture (see 18-750, below). In addition, microprocessors would be strongly introduced. This is reasonable since microprocessors involve subject matter related to both logic design and computers, which is the material the course is intended to cover. Finally, consistent with the above microprogramming is also introduced with more emphasis in 18-234.

* 18-743 Computer Organization and Real Time Processing: Remains the same.
* 18-745 Minicomputer Systems: Here, students could also engage in microprocessor projects if they wish. Thus, 18-745 becomes the course in which advanced microprocessor concepts are learned. If the student performs a microprocessor project, 18-234 is the prerequisite. Otherwise, 18-233 and 18-743 are prerequisites, unless waived by the instructor.
* 18-750 Advanced Computer Structures: This course will remain the advanced computer architecture course. Logic design will not be covered here since it is contained in 18-234.
* 18-859 Advanced Topics in Computer Engineering: Remains the same.

- Introduction of Microprocessor Topics into Specific Individual Courses:
  * A good example of this approach was the microprocessor-related digital control problem assigned to the students of the Senior Fundamentals of Control course, 18-301 in Fall, 1975. The objective was to design a microprocessor-based automotive fuel-injection control system. Since this problem is a real one which has not as yet been technically or economically solved, it was received with great interest. A copy of the problem statement is given in Appendix A. The solution was to be a "paper" design with transducers, control means, and economics to be considered.
  * The third course in a sequence of Analysen, Synthesis, and Evaluation (ASE) courses was offered for the first time in Spring, 1976. This course, ASE III, is a project course involving Junior and Senior students of all engineering disciplines working under a unifying theme. For the Spring, 1976, the course was entitled "Microprocessors in Engineering." Under this theme, five projects with an average of five students each were conducted. These projects were:
    - Design of a microprocessor-controlled solar thermal heating and cooling system;
    - Design of a portable, microprocessor-based gas detector utilizing solid-state gas sensors;
    - Design of a microprocessor-based traffic-light controller and traffic simulator;
    - Design of a microprocessor-controlled foundry cleaning machine;
    - Design of microprocessor-controlled dual mode bus transportation system.

  It is to be noted that each of the above projects included design, construction, and testing of a microcomputer from the chip level and incorporation of the microcomputer into the individual project design.

- Development of New Courses:
  * In order to meet the needs of the non-Electrical Engineering undergraduate/graduate students, a new course, 18-235, entitled "Topics in Computer Engineering - Microprocessors" was offered for the first time in the Summer Session of this year. The course was designed to provide a basic understanding of microprocessors and microcomputer systems. Topics such as programmed logic, semiconductor technologies, memories, architecture, software, interfacing, and high level development systems were covered. Laboratory sessions for the course were conducted utilizing microcomputer work stations, with high level development systems, and cross-assemblers and simulators on a host PDP-11/40.
  * An intensive five-day summer course for industry, entitled "Microprocessors - Phase II", will be conducted for the first time in July, 1976. This course was developed in conjunction with industrial research, development, and manufacturing managers. It is aimed at going beyond the introductory courses by providing the attendee with the opportunity to study, evaluate, design, and apply microcomputer system solutions to relevant industrial problems utilizing equipment of different manufacturers. The course is divided into lectures, "hands-on" workshop sessions, and panel discussions. In the workshop sessions, the class will be organized into small, creative teams. Each team will be presented with a different design problem for which they must then proceed to develop a hardware/software solution. The design solution associated with each project will be presented by the respective groups in order to share experience gained and to reinforce design concepts. A preliminary description and outline of the five-day industrial course is presented in Appendix B.

- Student Projects:
  Undergraduate and graduate students can obtain credit for working on a project under the supervision of a faculty member. A number of students have successfully undertaken these projects in the microprocessor area. Some typical projects have been implementation of Synchronous Data Link Control (SDLC) on an LSI-11 Microcomputer, Parallel Microcomputers Utilizing Different Microprocessors, Dynamic Memory Allocation for a Microcomputer, and Design of a General-Purpose Microcomputer with Special Features and Interfaces. The latter project was undertaken by a Freshman Electrical Engineering Student and a summary of this project is given in Appendix C.

Hardware

Hardware capability for the various courses was incrementally acquired with decisions on equipment being made on the rapidly changing technical and economic merits of microcomputers of different manufacturers. The hardware presently being utilized and reasons for its selection are as follows:

- MOS-Technology KIM-1 Microcomputer System on a Board (10 units): This system is a completely assembled card which includes a keyboard, 6-digit display, interface to a teletype and audio cassette, 2K ROM with monitor firmware, 30 input/output pins, 2 timers, crystal clock, 1K user Random Access Memory, and three detailed manuals. This unit lists for $245 in small quantities and is the basis for our individual or small group laboratory work stations. These units will be used with a higher level development system and with a PDP-11 for software and hardware debugging.
- MOS Technology MDTS Higher Level Development

System: This system permits assembly, editing, and debugging of the microcomputer software and provides debugging capability for the hardware.

- Intel MDS-800 Intellec MDS Microcomputer Development System: This development system is used mainly with projects designing microcomputers at the chip level and provides assembly, editing, and debugging capability for our 8080-based microcomputer projects.
- Intel Intellec 80 Microcomputer Development System: Provides capability similar to MDS-800 for our 8080 chip design projects.
- RCA Cosmac Microkit Development System: This unit is a higher level development system in rack-mounted form which gives us debugging and program evaluation capability for the RCA CMOS microprocessor.

Software

Both resident and cross-software are utilized in the microprocessor courses. Resident monitor routines on the KIM-1 laboratory work stations and on the development systems allow the student to interact immediately with a microcomputer and obtain an immediate feel for the device. The monitor program in the microcomputer laboratory work stations controls the interface to an ordinary audio cassette recorder so that the student, again, can immediately write and retrieve records from the recorder. The monitors in all the systems provide for such functions as examination and modification of storage locations, movement of data to or from storage locations, single-step debugging, breakpoint setting, reading and punching paper tape in particular formats, and PROM programming. In addition, resident assembler and editor programs are available on the development systems. Unless high speed means of entry of these programs into the system are available, however, their usage can be tedious owing to the long time (approximately 30 minutes each) required to load them into microcomputer memory. Since memory space is generally limited, entry of different software into the system displaces previously entered software. Thus, an editor program would be replaced by an assembler which then would be replaced by the program to be executed. Another assembly, then would require reloading of the assembler. Some development systems offer these programs permanently mounted in ROM and, therefore, they are continuously available to the user without external loading.

In order to alleviate this problem, the MDS-800 Development System (which is in the Mechanical Engineering Department) was interfaced to a PDP-11/40 Minicomputer. This allows the microcomputer assembler and editor software plus other programs to be stored on the minicomputer disk storage and loaded directly upon call to the microcomputer memory for execution. This capability permits interesting laboratory exercises in minicomputer/microcomputer-based control systems.

In addition to resident software and standard cross-software supplied by the microprocessor manufacturers, a series of cross-assembler and simulator programs for microcomputers were developed at CMU to run on a host PDP-11/40 minicomputer. These programs were developed by a Ph.D. candidate, Mr. Thomas Dickey, and are written in modular fashion incorporating a large percentage of code which is common to assemblers and simulators in general, regardless of which particular microprocessor is being used. With this basis of code, an assembler and simulator for any microprocessor can be written in a very short time. At the moment, assemblers and simulators for nine microprocessors are available on the PDP-11/40.

Conclusions

The interdisciplinary nature of microprocessor system design and application and the correspondingly different levels of instruction necessary lead to multiple approaches to microprocessor education. At Carnegie-Mellon, a mixture of redesigned computer engineering courses, interdisciplinary engineering courses, new courses for non-engineering students and for industrial professionals, course term problems, and student projects provide the opportunity for microprocessor education to all who desire it and at a level corresponding to their backgrounds and requirements.

It has been found that except for specific projects, a microcomputer already assembled and tested on a card level is better suited for laboratory or hands on exercises than building a microcomputer from scratch with chips. The latter approach results in an extraordinarily long time to completion because of soldering or wire-wrap inexperience and lengthy and ill-defined hardware and software debugging techniques. The assembled and tested card allows immediate application of concepts but still permits architecture study. An intermediate solution is the printed card which requires insertion of supplied sockets and chips and also includes a ROM monitor program.

Also, software assembly and simulation on a minicomputer used in conjunction with high level development systems provide fast and flexible software and hardware design verification and debugging means.

Experiences with the courses conducted to date yield the following observations:

- Unless deliberately and constantly monitored, interdisciplinary engineering students in a microprocessor-based project course will gravitate toward their own discipline if possible, and will not gain the amount of microprocessor experience potentially obtainable from the project;
- Stressing of chip technology (PMOS, CMOS, etc.) and ramifications of the technology on circuit density, speed, cost, and application is an important portion of course content and contributes to better understanding of microcomputer operations and limitations;
- Standard higher level language compilers for different microprocessors would be very desirable, especially for interdisciplinary engineering and scientific applications;
- The still relatively high cost of peripheral devices such as A/D converters and transducers are incompatible with microprocessor costs and limit the variety of exercises which can be implemented for student education.

As microprocessor mature, some standardization in software and hardware may develop and thereby make the computer-on-a-chip a truly universal building block and main element of a family which in cost and ease of application is really the alternative to random logic.

Appendix A

Digital Problem EE 18-301 R. L. Krutz

Objective: Design a digital controller for an automotive fuel injection system.

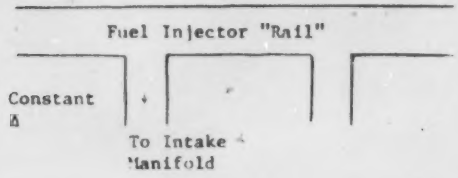
Method: Control the air-fuel ratio (mass).

Background: In order to control the air-fuel mass ratio, the mass of air and mass of fuel injected into the engine must be determined.

Air mass determination - The automobile engine, which is itself a positive displacement device, can be used as an air mass measurement system. If the instantaneous values of the following parameters are known, the mass of air through the engine (positive displacement meter) can be determined.

* Engine speed (RPM)
* Manifold vacuum & temperature
* Water temperature
* Ambient temperature
* Throttle position
* Volumetric efficiency (measure of engine's ability to "breathe")
* Barometric pressure (assume constant)
* Engine displacement (bore x stroke)

Fuel mass measurement - Assuming a constant pressure from fuel injector "rail" (supply) to intake manifold and knowing volume of port into intake manifold, fuel temperature, and fuel specific gravity (=.86), the mass of fuel into the cylinder can be determined by the time the injector is open.



Problem Statement:
- Develop an algorithm to acquire required data, process the data, and control the air mass/fuel mass ratio.
- Implement the system with a microprocessor or a hardwired logic system.

Pertinent Information:
- Air Mass ≡ K · (engine speed) · (manifold vacuum) · (manifold temperature) · (water temperature) · (ambient temperature) · (throttle position) · (volumetric efficiency) · (barometric pressure (constant)) · (engine displacement (bore x stroke)).
- Air Mass/Cylinder ≡ above expression ÷ 8 (assuming 8 cylinders).
- Fuel Mass into Cylinder ≡ R · (pressure drop from "rail" to intake manifold) · (fuel temperature) · (fuel specific gravity) · (time injector is open).
- Control will be implemented by controlling time injector is open.
- Values of volumetric efficiency vs. engine rpm to be used in problem

RPM	Volumetric Efficiency (%)
400	82
1200	80
2000	77
3000	71
3600	63

- Some typical values of parameters
  * Engine speed - idle - 400 RPM
    60 mph - 3000 RPM
  * Vacuum - 1 atmosphere (high speed)
    27 inches high (low speed)
  * Throttle angular position -
    0° (closed throttle)
    70° (high speed)
  * Air Mass/Fuel Mass 12#/1# (max power)
    15#/1# (max economy)

Requirements: Detailed written explanation of problem solution including:
* Transducer discussion
* Flow charts
* Assumptions
* Calculations
* Program (if any)
* Schematic diagram (detailed)

Problem solution should be typewritten or neatly printed and under a cover.

References: Automotive Engineering, "Review of Automotive Electronics," Nov. 1974, Vol. 32, #10, P. 29.
Automotive Engineering, "Closed Loop Feedback for Engine Self-Tuning," March, 1975, Vol. 84, #3, P. 27.
Automotive Engineering, "Solid State Devices for Automotive Electronics," April, 1974, Vol. 82, #4, P. 32.
Proceedings - Convergence 74, SAE, October 28-30, 1974, Troy, Michigan.

Appendix B

Industrial Course: Microcomputers - Phase II

I. INTRODUCTION
- Programmed Logic Concepts
- Relation of Microprocessors to Other Computers
- Areas of Application

II. TECHNOLOGY
- PMOS
- NMOS
- Bipolar
- I^2L
- CMOS
- Characteristics of Different Technologies
- Trade-offs
- Advantages and Disadvantages

III. BUILDING BLOCKS
- Registers
- Counters
- Arithmetic Logic Units (ALU)

IV. ARCHITECTURE
- ALU Control
- Data Paths
- Addressing
- Timing
- Machine States
- Options
- Microprogrammable Control
- CPU Chip

V. SOFTWARE

- Machine Representation of Information
- Instruction and Data
- Machine Language
- Assembly Language and Assemblers
- Higher Level Compilers
- Editors and Simulators
- In-class Program Examples

VI. MEMORIES

- Types
- Methods of Data Storage
- Organization
- Addressing
- Decoding
- Access and Cycle Times
- Power Requirements
- Memory Arrays

VII. INPUT/OUTPUT

- Objectives
- Classes of I/O Devices
- Corresponding I/O Structures
- Timing Considerations
- Peripheral I/O Chips
- Software Considerations
- Economics
- In-Class Development of I/O Hardware and Software Structures

VIII. MICROPROCESSOR EVALUATION TECHNIQUES

- Definition of Tasks
- Classification of Programs
- Instruction Set Evaluation
- Memory Size Evaluation
- Speed Evaluation
- Benchmark Tests
- Microprocessor vs. Random Logic Implementation
- Microprocessor vs. Custom LSI Implementation
- Second Sourcing
- Support

IX. ECONOMICS

- Programmed vs. Random Logic Costs
- Software Costs
- System Costs
- Trade-offs
- System Cost Estimation Techniques

X. DETAILED MICROCOMPUTER DESIGN FOR INDUSTRIAL APPLICATIONS

- Chip Interconnections
- CPU Chip
- Memory Chips
- Clock Chips
- Input/Output Chips
- Timing
- Interfaces
  - Multiplexers
  - A/D Converters
  - D/A Converters
  - Direct Digital Input
  - Interrupts
  - Connection to External Devices
- Software
  - Algorithm
  - Flow charts
  - Necessary Routines
    - Data Acquisition
    - Input/Output
    - Counting
    - Noise-Rejection
    - Delay Generation
    - Computation
  - Writing Complete Program
  - Assembly and Editing of Program
  - Debugging Techniques
  - Demonstration of Program

XI. OTHER APPLICATION DISCUSSIONS

- Process Control
- Environmental Monitoring
- Manufacturing Automation
- Point-of-Sale
- Data Communications

XII. CLASS DESIGN PROJECT

- Problem Description
- Design Objectives
- Hardware Design
- Software Design
- Implementation of Microprocessor Development
- Performance Evaluation
- Documentation

XIII. PANEL DISCUSSION AND OPEN FORUM

- (Invited Authorities Will Answer Applications Questions)

XIV. REVIEW AND CRITIQUE

Appendix C

MICROCOMPUTER 5 (M5): A Freshman Design Project

Guy Sotomayor

M5 (Microcomputer 5) is a general purpose computer utilizing an 8080A microprocessor. The design was implemented using 7400 series TTL for reliability and flexibility. It was designed so that system expansion would be easy to implement.

The main reason for designing M5 was to investigate the usefulness of moving the push-down stack out of main memory into a physically separate memory. The stack bit status word could be used as the chip enable for any memory used for the stack.

Another reason for M5's conception was to encode the date in octal format and present it on seven segment panel displays. Only after construction was completed was it realized that several alternatives to the encoding scheme were possible which had distinct advantages over the random logic implementation used.

The organization of M5 is primarily implemented with three buses. They are - the data bus, address bus, and control bus. By designing M5 in this manner, it became very easy to expand the system. Also, with this bus structure, debugging was very easy. From the time M5 was completed until the first sample programs were running, a little over five weeks had elapsed. A good portion of the time was spend finding bad connections and poor sockets. Some time was spent correcting five logic errors which had gone unnoticed until the debugging had started.

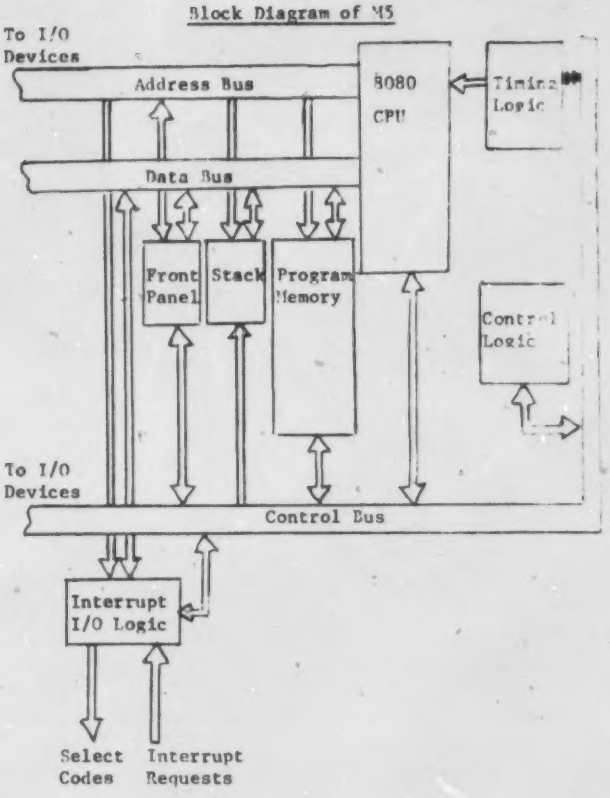
M5 presently resides on four boards. The first and largest board contains the CPU, stack memory, front panel logic, and the timing and control circuitry of M5. The first of three smaller boards accomodates up to 3K bytes of RO, and 1K bytes of RAM. The second board contains 12K bytes of dynamic memory and the required refresh circuitry. The third and final board contains the I/O and interrupt logic for M5. It also contains an interface to an IBM 1052 Selectric Terminal, which is the primary I/O device for the system. Within the next two months, it will also contain an interface to an audio cassette deck for mass storage.

At present the only software written for M5 are several demonstration programs and a monitor which resides in 2K of memory. Hopefully most programs written for other 8080-based systems can be adapted to M5. The only programs that will require any conversion would be those that perform I/O or use the stack in unconventional ways.

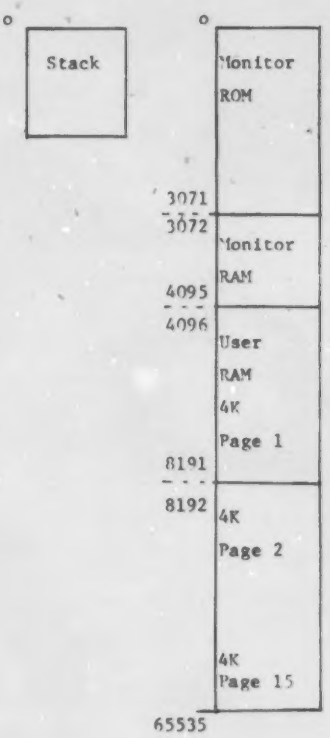
Several items are planned for M5 in the future. A hardware floating point arithmetic unit is planned so that computations can be performed rapidly. Then, additional processors will be added to the system to perform I/O and various other time-consuming functions. Finally, M5 may eventually be used as a front-end to a micro-programmable processor with writable control store.

A system block diagram, memory organization chart, and monitor listing follow.

Block Diagram of M5



Memory Organization of M5



M5 Monitor Commands

Command	Action Taken by Monitor
CA	Cancel current program execution
CB #	Clear Breakpoint Register, #
CO	Continue program execution
DB [#]	Display Breakpoint Register, #
DM LADDR, HIADR	Display Memory from LADDR to HIADDR in 16 byte increments
DR [#]	Display 8080 Register, #
LM STADR	Load Memory starting at STADR. Data is entered after crlf , successive bytes entered by putting a comma between data. crlf 's may be inserted freely. To exit type 'X crlf .
LR # = BYTE	Load 8080 Register, #, with BYTE
MV LADR, HADR = NADR	Move data between LADR and HADR to memory starting at NADR.
PA	Pause current program
RE	Restart the monitor
GO STADR	Start program execution at STADR

[#] - indicates that the argument may be left off and all registers will be displayed.

DESIGN WITH MICROPROCESSORS IN AN INTERDISCIPLINARY ENGINEERING PROJECT COURSE

Drs. Baumann, Fenves, Feucht, Krutz, Murphy, Paul, Tuma, and Sauer
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Abstract

Microprocessors have been used as the unifying theme in an inter-departmental design course. Under the theme of "Microprocessors in Engineering" student groups undertook design efforts on five separate projects. Students from the Electrical Engineering, Mechanical Engineering, Civil Engineering, and Physics Departments formed interdisciplinary groups to work on the projects.

To introduce the fundamentals and workings of microprocessor systems to the heterogeneous student body taking the course, a three-day crash period was used to expose the students to several microprocessor systems and to give them hands-on experience. A wide variety of literature was made available to the students to further delve into the area on their own.

The students have attempted difficult and sophisticated projects using microprocessors. Their final design work in hardware and software has met the project objectives.

Introduction

A three course sequence in analysis, synthesis and evaluation designated ASE I, ASE II, and ASE III, has been instituted at Carnegie Institute of Technology of Carnegie-Mellon University, to provide a problem-solving experience to undergraduates extending from their Freshman year until they graduate. These courses provide a new emphasis on the Carnegie Plan of Professional Education which has been the educational philosophy of Carnegie-Mellon University for over thirty years.

The focus of the ASE courses is on the problem and its treatment to obtain a solution compatible with economic and social constraints. The ASE courses are designed to challenge the student to seek and utilize whatever principles or knowledge are applicable to the solution of the problem.[1]

The final course, ASE III, in the three-course sequence consists of a project designed to stress the solution of multidisciplinary professional engineering problems. ASE III is interdepartmental, asking students to utilize what they have previously mastered to solve a relevant engineering design problem. The students work as a team in groups of four or five on a central theme. The objectives of this course are:

1. To have the students acquire an in-depth problem-solving experience in a multidisciplinary project that emphasizes aspects of engineering analysis, synthesis and evaluation.
2. To have the students apply a multidisciplinary approach to the solution of engineering problems and to develop the ability to coordinate their various fortes to bear on a complex engineering problem.
3. To provide the students with an educational experience which reflects a microcosm of real life engineering problems, including the development of hardware, where applicable.

A problem theme is selected by a course committee composed of faculty and students during the semester prior to the conduct of the course. In the Fall 1975 Semester, the committee chose "Microprocessors in Engineering" as the theme for the Spring 1976 offering of ASE III.

Projects and Teams

To insure that the projects were relevant to actual engineering problems, an effort was made to identify project objectives with research interests of faculty supervising the course. Along the same vein, the role of microprocessors was made an integral part of each project, not an embellishment.

Five projects under the theme of Microprocessors in Engineering were identified and presented to junior and senior students in the form of rather general statements, which are presented here in abbreviated form:

1. A Solar Energy Heating and Cooling System
   Design, build and test a solar-powered heating and cooling system operating under the control of a microcomputer.
2. A Portable Monitor for Explosive Gases
   Design, build and test a microcomputer-controlled portable solid state gas detector system to monitor the concentration of methane and carbon monoxide gases in a coal mine environment.
3. A Microprocessor-Based Traffic Light Controller
   Design, build and test a prototype microprocessor-based traffic controller that optimizes traffic flow at a busy intersection.
4. Microprocessor Control of Automated Foundry Cleaning Machines
   Design, build and evaluate a microprocessor-based controller to position an existing "proof-of-concept" foundry cleaning machine in a "learn and repeat" mode of control.
5. Microprocessor Control of a Dual Mode Transportation System
   Design, build and evaluate a microcomputer-based system for controlling a model dual mode test track for an automated transportation system.

Student teams had to take one of these project statements and use it as the basis for the formulation of definite tasks to be undertaken to meet a set of objectives also defined by the student teams. Faculty involved in the course were to act as resource and management people, and were not to be involved in the daily workings of the student teams. Twenty-seven students -- seventeen from electrical engineering, five from mechanical engineering, four from civil engineering, and one from physics -- registered for the course. Eleven students were in their senior year, the others in their junior year. An effort was made to provide a student mix for the teams - with each drawing students from at least two departments. In addition, a mix of interests and capabilities in hardware and software was arranged in the make-up of the teams.

Since the students registered in the course -- and most of the faculty managing the projects — had no working experience with microprocessors, it was decided to hold a three-day crash course during the first week of the semester to introduce the fundamentals of microprocessors to the class.

Crash Course on Microprocessors

The course was designed to accomplish the following

objectives:

- familiarize participants with areas of application of the microprocessor and its relation to other computers;
- provide a basis for selection and evaluation of microprocessors for a particular design problem;
- impart an understanding of the microprocessor architecture so that in conjunction with memory and other peripheral chips an efficient microcomputer and appropriate interfaces could be designed and implemented;
- develop basic programming skills so that students and advising faculty could write, assemble, and edit microprocessor code on both a microcomputer development system and a PDP11/40 minicomputer.

Because of the interdisciplinary nature of the students' backgrounds, the initial lectures for the first half-day of the course were concerned with developing a common basis of supporting knowledge for subsequent lectures on chip technology, memories, microprocessor architecture, and software. This basis included a brief review of number systems and digital subsystems on the level of counters, registers, arithmetic operators, and arithmetic logic units.

Following the introductory topics, a lecture on chip technology was presented. This material is considered important since terms relating to chip technology continually arise in discussions of memories, microprocessor chips, and interface chips. In addition, an understanding of the speed, cost, density, power, noise immunity, and reliability implications of the various technologies is vital to the intelligent selection and evaluation of microprocessors.

The last prerequisite for microprocessor architecture discussion -- memories -- was then covered. Cell structures of RAMS, ROMS, PROMS, and EPROMS were presented along with typical organizations illustrated by popular memory chips, addressing, and memory expansion techniques. A major discussion of microprocessor architecture then commenced, beginning with typical microprocessor organization and followed by discussions of the Motorola 6800, Intel 4040, RCA COSMAC, and the Intel 8080. The Intel 8080 was used for in-depth discussion of microprocessor organization, operation, and software for the remainder of the course. Detailed supporting manuals, specification sheets, application notes, and supplementary notes on the 8080 microprocessor were provided. Examples of interfacing, control signal generation, and Direct Memory Access operations were given.

The software section of the course again used the 8080 as the means of presentation. The 8080 machine language and assembly language were covered in depth and homework problems requiring coding in both languages was given. In addition, material on an 8080 cross assembler and editor for a host PDP 11/40 was presented and "hands-on" demonstrations were conducted. Demonstrations on the RCA COSMAC Microkit were also conducted to illustrate its architecture and special features. The microprocessor course outline is given in Appendix I.

Results of the Project

In a very short time, the students became proficient in the use of microprocessors as design elements. It became apparent that the implementation of the project tasks, rather than the microprocessor hardware and software, became the major objective. The rapidity and the confidence with which the students were able to work with the microprocessors and peripheral circuitry was very surprising, in view of the warnings that have been voiced in the literature as to the difficulties that digital design engineers would face when using microprocessors.

Faculty supervisors acted throughout the course according to plans: as managers, resource people, and expediters. Student teams met with their supervisors once or twice a week to review their progress, to receive feedback, and to plan for further work. A natural selection process allowed each team to assign its own leader, who made sure that tasks were completed, and coordinated his team's activities.

To provide checkpoints for the progress of the teams, two intermediate presentations of student work were made. The presentations were attended by all the students and faculty involved in the course. The interaction among the participants provided a very useful input to the teams.

A director for the course coordinated various aspects of the teams' activities. This involved processing purchase orders, providing equipment, scheduling presentations, supervising budgets, and arranging final report preparations.

At the end of the semester, the project teams made a public presentation of the results of their work. In addition, each group submitted a technical report summarizing the design, the construction of hardware, and an evaluation of the final product. The abstracts for the five reports follow:

1. A Solar Heating and Cooling System

   The objective of the Solar Energy Group was to design a Solar Experimental Facility to be constructed on Campus which would have the capability to run experiments on solar thermal energy collection for heating and cooling of buildings. The Solar laboratory will incorporate a data acquisition system, designed to obtain data regarding the amount of energy that can be collected and the space that can be heated and cooled given a spectrum of weather conditions. In addition, the laboratory will perform an emulation function where a range of building sizes can be simulated using specific collector and storage configurations.

   A microcomputer system will be used to perform the data acquisition, data manipulation control, and specific system emulation.

   As a proof of concept, a model was constructed utilizing a flat plate collector heating a 4-foot cube "space." This model allows evaluation of the concepts required to implement the Solar Experimental Facility design.

2. A Portable Monitor for Explosive Gases

   The primary goal of this work is to develop a portable instrument utilizing solid state gas detectors which can accurately determine concentrations of methane and of carbon monoxide in the presence of other gases such as might be found in a coal mine environment. Solid state detectors are of particular interest because they can provide substantial improvement over more traditional measurement techniques in terms of size, speed, and cost. However, currently available solid state detectors are characterized by poor selectivity and widely varying sensitivities. This project attempts to compensate for these deficiencies by utilizing an array of TGS detectors with a microcomputer to control relative sensitivities and to process the data thus obtained in a meaningful manner. Signal processing and

electrical inputs to the detectors are used to eliminate individual detector variations, and to improve the selectivity of the system. With proper detector excitation, the microcomputer can compare responses at different sensitivities to determine the concentration for a given gas with far more accuracy than could be obtained from static testing. Since it is also possible to obtain different sensitivities to different gases, there is a capability for discriminating between arbitrary mixtures of gas. Some recommendations are provided to aid in the development of better solid state detectors for gas concentration measurements.

3. Microprocessor Controlled Traffic Light
With the advent of sophisticated computer power in the cheap and easy-to-use form of microprocessors, highly intelligent traffic controllers can be built for the management of intersections. This project explores the management requirements for a traffic intersection. The simulation of an intersection, and the testing of hardware, software, and management techniques are also explored together with an investigation of the marketability of a microprocessor-based traffic controller in the present market.

4. Microprocessor Control of Automated Foundry Cleaning Machines
In order to improve batch part manufacturing productivity, foundries have to introduce automated machines into their cleaning rooms. A "proof-of-concept" steel casting finishing machine is under development at Carnegie-Mellon University, consisting of a modified Sears and Roebuck radial arm-saw with hydraulic drives. The purpose of this project is to design and implement a microcomputer in conjunction with attendant computer-machine interfaces to provide the machine with "learn and repeat" capabilities. The project was divided into three basic areas. Hardware included the actual design and building of the micro-computer with A/D and D/A interfaces. In addition a study of the machine system was completed to determine the transfer function which defines the process dynamics. The software development involved writing the control program, including the input/output routines and the 16-bit calculations of the control variables for closed-loop microprocessor control. The control aspect involved design of a digital controller using discrete control theory, a mathematical model of the physical system, and the desired performance specifications. The specifications included table positioning within an accuracy of + 1/32 of an inch with the velocity of the positioning axis of at least 1 inch per second.

5. Microprocessor Control of a Dual Mode Transportation System
The Dual Mode Vehicle System is an experimental mass transit system that allows a variety of vehicles to be operated under either driver control or automatic electronic control. The transformation from standard vehicle to Dual Mode Vehicle is accomplished by adding an electric traction motor for motive power on the guideway, two guidance arms for steering and power pickup, and an electronic control system for motor, brake, and guidance control. On regular roads and highways the vehicle is controlled completely by the driver; on the system guideway, it is under automatic control. Both the system guideway computer and the vehicle on-board computer are configured from microprocessors.
The guideway controller is responsible for routing vehicles, billing, and guideway safety. The on-board vehicle controller is responsible for motor control, brake control, and guidance arm control. The guideway controller and the on-board vehicle controller communicate over the guideway power rail and through message locations along the guideway. Microprocessors can deal rapidly with the quantity of information needed for safe operation of the vehicles on the guideway.

Conclusion

Interdisciplinary student teams have successfully worked on design projects using microprocessors as a unifying theme. Using a three-day crash period, students were introduced to the fundamentals of microprocessors and digital hardware and software. In the process of accomplishing tasks needed to meet their project objectives, students had to further educate themselves in designing with microprocessors. A working proficiency in microcomputer systems was thus acquired to the extent that they were capable of using microcomputers as part of their project design work.

Students divided the tasks for the projects according to ability and interest. Electrical engineering students tended to monopolize the hardware aspects of the microcomputer design; but most students got involved in the software aspects. However, there were instances in which non-electrical engineering students took the leadership in the hardware area.

Throughout the conduct of the course, certain problems arose that had to be resolved. Some projects started out too ambitiously; redefinition of the objectives and tasks brought back the project scope to within the reach of the students. Project demands on student time were sometimes excessive and competed with the needs of other courses; students resolved this difficulty according to their priorities and interests. There were also the usual procrastinators who did not contribute their fair share to the team effort; these students were reminded of their responsibilities and of the effect of their inaction on their course grade.

Our experiences in this course lead us to believe that we have identified an effective approach — the project format — to providing a large number of students with a working knowledge in the area of microprocessors at various levels ranging from actual design of microprocessors hardware and software to higher-level design of systems in which a microprocessor is an integral component.

The project course on "Microprocessors in Engineering" was supported by a grant from the Corning Glass Work Foundation.

References

1. C. P. Neuman, D. T. Tuma, D. L. Feucht, F. W. Paul, and B. R. Teare, "Analysis, Synthesis, and Evaluation, Adventures in Professional Engineering Problem Solving," Frontiers in Education Conference, Atlanta, Georgia, October 1975.

Appendix I

Outline for Microprocessor Crash Course

I. INTRODUCTION

- Programmed Logic Concepts
- Relation of Microprocessor to Other Computers
- Areas of Application

II. MICROPROCESSOR CAPABILITIES

- Functions Performed
- General Limitations

III. NUMBER SYSTEMS AND DIGITAL SUBSYSTEMS

- Number Systems
- Digital Subsystems
- Registers
- Counters
- Arithmetic Operators and Circuits

IV. TECHNOLOGY

- PMOS
- Bipolar
- NMOS
- CMOS
- I^2L
- Characteristics of Different Technologies

V. MEMORIES

- Types
- Organization
- Addressing
- Access and Cycle Times

VI. ARCHITECTURE

- Microprocessor Organization
- Typical Microprocessors
- Intel 8080
- Timing
- Instruction Cycle
- Addressing
- Instructions and Data
- Instruction Execution
- Stacks
- Input/Output

VII. SOFTWARE

- Machine Representation of Information
- Machine Language
- Assembly Language and Assemblers
- Higher Level Compilers
- Editors and Simulators
- Algorithms
- Flow Chart
- In-Class Examples

VIII. MICROPROCESSOR DEVELOPMENT SYSTEMS

- Purpose
- Description
- Features
- Typical Application

IX. OTHER PERIPHERAL CHIPS

- Clock Chips
- I/O Ports
- A/D Converters
- D/A Converters

X. IN-CLASS PROBLEM AND DEMONSTRATION

XI. REVIEW AND CRITIQUE

MICROCOMPUTERS IN THE COMPUTER ENGINEERING CURRICULUM

John F. Wakerly and Edward J. McCluskey

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California

1. INTRODUCTION

Microcomputers are the most important new building block for electronic systems to emerge since the integrated circuit (IC) operational amplifier. It is obvious that colleges and universities should incorporate microcomputers into existing electrical and computer engineering curricula; but not surprisingly, there exists some dispute about the best way to do this. Before describing the approach at Stanford, we will discuss the philosophy upon which our curriculum is based.

2. PHILOSOPHY

The word "microcomputer" means many things to different people. A microcomputer is best characterized by a phrase that depends on one's point of view:

MOS LSI processor
Single-chip processor
8-bit processor
Few-chip computer
Dedicated computer
Personal computer
Small computer
Inexpensive computer

The first seven phrases above are, for the most part, valid characterizations of current microcomputer technology and applications. However, the "microcomputer revolution" is best understood by realizing that a microcomputer is basically just an inexpensive computer. The driving force behind the explosion of microcomputer technology and applications is economic [Electronics, 1976a]. The existence of inexpensive microcomputers affects product design in two ways. First, the low cost of microcomputers sometimes makes it possible to replace or simplify mechanical and analog devices by using a digital approach. An example of this type of application is an electronic cash register. Second, in applications where mechanical and analog devices are still more inexpensive than microcomputers, microcomputers are often used anyway; with the microcomputer approach many features can be added whose cost would be prohibitive with a conventional approach, and the overall cost per feature of the product becomes lower. Intelligent test instruments are examples of this type of application. As the low cost of microcomputers causes their proliferation, demand further lowers their cost and spurs on the manufacturers to develop even better microcomputers.

After characterizing microcomputers above as inexpensive computers, we dwelled on the word inexpensive. But from the educational point of view, the important aspect is that microcomputers are just computers. There is nothing essentially different between microcomputer and minicomputer architecture, just as there was nothing essentially different between minicomputer and maxicomputer architectures ten years ago. (The important aspect of the "minicomputer revolution" was also a proliferation of applications created by economic factors.) Thus, institutions which already have strong digital systems or computer engineering programs will have no problem teaching students about microcomputers. We reinforce this view later by describing the concepts that must be learned to become a proficient user of microcomputers, and then showing that most of the concepts are already covered in existing curricula.

Universities that do not already have strong digital systems curricula may have some problems teaching students about microcomputers. In the absence of an existing curriculum, one might attempt to develop a course that teaches only the background material relevant to microcomputers and then plunges into microcomputer specifics. We feel that this is a mistake. The microcomputer revolution is just an indicator of the growing importance of digital electronics, signaling the need for improved digital systems curricula at the undergraduate level. Rather than putting together a "quickie" microcomputer course, a developing department should concentrate on providing a suitable framework of undergraduate digital systems courses from which specialization in microcomputers or any other important topic easily follows.

3. MICROPROCESSORS AND MICROCOMPUTERS

Before continuing, we should clarify the terms "microprocessor" and "microcomputer." A microprocessor is a central processing unit (CPU) on a single LSI chip. A microcomputer is a microprocessor combined with memory and input/output (I/O). Sometimes a processor built from multiple LSI circuits, such as bit slices, is called a microprocessor. This usage is acceptable though uncommon. The crucial aspect of the definitions is that a microprocessor is a CPU only,

whereas a microcomputer includes memory and I/O [Mano, 1976].

Microprocessors consisting of a CPU on a single LSI chip are the leaders in current technology and applications. Processors with 8-bit wide data paths are the most common, but 12-bit and 16-bit processors have appeared and more will be announced in the near future. Also soon to appear are single-chip microcomputers with read-only memory, read/write RAM, and I/O facilities, all contained in the same LSI package with a CPU [Electronics, 1976a].

Naturally, our discussion of microcomputer education includes microprocessors, just as computer design includes CPU design.

4. MICROCOMPUTER CONCEPTS

In this section we discuss several concepts that must be studied by potential microcomputer designers and users. We describe the similarities and differences between each concept and concepts needed by minicomputer designers and users.

Processor Organization

Just as early minicomputers were single-accumulator, single address machines, so were the first microprocessors. Common features of minicomputers are all found in current microprocessor architectures: index registers, multiple addressing modes (indirect, immediate, relative, auto-increment, etc.), stack pointers and stacked subroutine return addresses, block move instructions, and bit test instructions. Current microprocessors still lack some of the features of the most advanced minicomputers, such as floating-point arithmetic, user/supervisor modes, and multiple copies of registers for fast context switching. However, an important observation is that microprocessor architectures have evolved along the same lines as minicomputer architectures and there is no indication that this will soon change. It is not likely that advanced concepts such as associative processing, logic in memory, residue arithmetic, and so on, will be needed to understand microprocessor architectures in the near future.

There are a small number of features of some existing microprocessors that have not appeared in previous minicomputer architectures. Among these are register bank switching necessitated by the small number of register selection bits in an eight bit instruction [Fairchild, 1976], and limited-depth, on-chip stacks for subroutine returns and interrupts [Signetics, 1975]. One of the most interesting new features is in the TMS 9900 microprocessor [Texas Instruments, 1975]. The TMS 9900 architecture has no processor general registers, only a "workspace pointer register" that points to a small area of main memory that can be accessed and manipulated like general registers.

System Organization, Memory, and Input/Output

All current microprocessors use a simple linear bus for communicating with both memory and I/O devices, just like many minicomputers (most notably the PDP-11 [DEC, 1973]). A typical microprocessor bus has 16 address lines, 8 data lines, and a number of control lines. Different methods are used to access I/O devices on the bus. Some microprocessors, such as the M6800 [Motorola, 1975], use memory-mapped input/output similar to the PDP-11 Unibus. Others, such as the 8080 [Intel, 1975a] and the 2650 [Signetics, 1975], use certain control lines to distinguish between memory and I/O operations; during I/O operations they use the address bus to transmit port numbers or I/O select codes similar to the HP-2100 series of minicomputers [Hewlett-Packard, 1972].

Almost all current microprocessors have provisions for a direct memory access (DMA) facility to allow transfers between devices and memory without processor intervention. Single or multiple level vectored or unvectored interrupts are also found on all microprocessors. Interrupts can be selectively masked in microprocessors with multiple levels. Some microprocessors save interrupt status in fixed memory locations, while others push the status onto a stack. DMA and all of these interrupt variations can be found in minicomputers.

Probably the most unusual aspect of some microprocessor system organizations not found in minicomputers is the use of a multiplexed system data and address bus. Early microprocessors such as the 8008 [Intel, 1973] transmitted instructions and addresses byte-serially over a single 8-bit system bus. Thus the 8008 required three clock cycles to transmit a 16-bit address and an 8-bit datum, and external circuits were needed to assemble the address and data. Current 16-bit microprocessors such as the LSI-11 [DEC, 1975], PACE [National, 1975] and the CP1600 [General Instruments, 1975] have a 16-bit system bus over which data and addresses are multiplexed. In all cases, multiplexing is used to reduce pin requirements in the LSI package.

A number of other microcomputer system features may be unfamiliar to minicomputer users. Among these are the general lack of traps for exceptional conditions such as referencing nonexistent memory, and the use of read-only memory (ROM) as well as read/write random-access memory (RAM). An aspect of some small microcomputer systems that may be especially disquieting to minicomputer users is incomplete decoding of memory and I/O addresses. For example, if a system designed for a dedicated application requires only 256 bytes of memory, then only the eight low-order memory address bits are decoded to save hardware; the memory responds to any memory reference regardless of the value of the high-order address bits (see [Motorola, 1975] for more examples of this practice).

Software

The software concepts needed to program microcomputers are not very much different from those for minicomputers. Much microcomputer programming is done in assembly language, and microcomputer assembly languages have constructs and features very similar to minicomputers and maxicomputers. A few high-level programming languages for microcomputers exist, and more are being developed. There are a number of microcomputer-based disk operating systems, and minicomputer-like systems programs such as text editors and debuggers all exist on microcomputer systems.

Naturally the state-of-the-art of microcomputer software is not as advanced as minicomputer or maxicomputer software. The existing high-level languages such as Fortran and Algol are not yet supported. Run-time diagnostics from compilers are not as good as what one expects from large systems, and on-line debugging systems (with the exception of in-circuit emulators) are not as good as minicomputer debugging packages. Nevertheless, anyone who has used the software of a minicomputer can easily learn to use the software of a microcomputer.

Microcomputer "cross" software products are a type of software not usually encountered by minicomputer programmers. The somewhat limited capacity of microcomputers for large batch processing operations (both in computing power and peripherals) has caused the creation of cross-assemblers and cross-compilers that can be used to develop microcomputer software on a larger host machine. The use of cross-products and the mechanics of down-loading programs from the host machine to the microcomputer are important but straightforward concepts to be learned by the microcomputer user.

Circuit Technology

The progress in MOS LSI microprocessor technology over the last several years has been quite rapid. While new IC processing technologies have made a great impact on IC chip design, at the external electrical and functional levels nothing has been introduced which is not already understood from older technologies. At the electrical level, rather than requiring microcomputer designers to learn a whole new set of design rules and device characteristics, technology advances have actually made system design easier. Almost all MOS microprocessors, memories, and other support chips now have inputs and outputs compatible with TTL, a well known and understood logic technology. The packaging of ICs into dual in-line packages (DIPs) and the interconnection of DIPs on printed circuit or wire-wrap boards has not changed. Indeed, it is far easier for a designer of circuits using TTL MSI and SSI to learn the electrical characteristics and constraints of new microprocessor circuits than to learn about established high-speed logic families such as ECL. At the functional level, the basic functions of a computer have not been changed. The difference brought by LSI microprocessor technology is that a processor, device interface, serial communications unit, interrupt controller, or DMA module can now be implemented with a few standard chips instead of an entire circuit board.

Of course, there are a number of important differences between MOS microprocessor technology and older technologies. Electrical concepts such as limited fanout, capacitive loading, and three-state busing of microprocessors are not always taught to TTL logic designers. At the functional level, the general emphasis in microprocessor system design is usually shifted from brute performance to the direction of elegance of system design, ease of system interfacing, and minimization of SSI and MSI logic by using programs and memory instead of logic.

Hardware and Software Debugging

Oscilloscopes, multimeters, and logic probes are useful tools for microcomputer hardware debugging that are quite familiar to TTL logic designers. Likewise, assembler and compiler error messages, run-time compiler diagnostics, and on-line debugging packages are microcomputer software debugging aids that have been with us since long before the microcomputer revoltuion. Thus, the basic tools of microcomputer debugging should present no difficulty to the student of microcomputer design.

Two new debugging tools, logic analyzers and in-circuit emulators, have evolved because of the unique aspects of microcomputer systems. Logic analyzers are useful because, from the hardware point of view, a microcomputer system is just a large synchronous sequential machine. A basic logic analyzer has a clock input and 4 to 16 data inputs. At each clock pulse, the data inputs are strobed and their logic values are stored in memory. The stored input words are then displayed on an oscilloscope screen, either as binary digits or in a timing-diagram format. From 16 to 256 or more words of memory are provided, so that a long record of data input words can be observed. Triggering circuits are usually provided so that storage of data is not begun until a selected input word is observed.

In-circuit emulators allow the hardware and software of a primitive microcomputer system to be tested and debugged at operational speeds using the resources of a sophisticated development system. An in-circuit emulator is designed for a specific microprocessor, say the 8080; it has a cable with a 40-pin plug that is inserted in a system-under-test in place of the 8080 processor that normally runs the system. At the other end of the cable is an 8080 processor with special "hooks" that allow program execution in the system-under-test to be traced and interrupted by a user at the console of a more sophisticated development system. Furthermore, resources of the development system such as memory and I/O ports can selectively be made available to the system-under-test.

New microcomputer designers should be given the opportunity to become familiar with both logic analyzers and in-circuit emulators.

Peripherals

Many of the peripheral devices found in microcomputer systems are very familiar to minicomputer users. Teletypes, CRT terminals, papertape readers and punches, card readers, printers, plotters, and floppy disks are all common. On the other hand, some devices unfamiliar to many computer engineers are quite common in microcomputer systems. These devices, such as mechanical sensors and actuators, digital/analog converters, and other analog circuits, appear because of the expanded range of applications for low-cost microcomputer systems.

A primary difference between peripherals for minicomputers and peripherals for microcomputers is in how the peripherals are controlled. Minicomputer peripherals typically have complex interfaces to minimize the CPU time for transfers, whereas in many microcomputer applications the peripherals are controlled at a very primitive level by the CPU. Hence a microcomputer programmer often needs to devote more attention to input/output programming than a minicomputer programmer.

Development Systems

Microcomputer development systems are a new type of system necessitated by the fact that a typical microcomputer system is incapable of supporting its own development and subsequent debugging and modification. For example, a microcomputer-based point-of-sale terminal or ping-pong game is hardly capable of supporting program text-editing or assembly. New minicomputers were typically "bootstrapped" into operation by writing cross-assemblers and/or compilers for a larger machine to get initial software into operation, and then using the new minicomputer system for subsequent debugging and development. New microcomputer systems are typically supported throughout their lifetimes by development systems having the assembly, compilation, text-editing, and file storage capabilities that most final microcomputer systems never have because of their limited applications. A typical microcomputer development system such as

Intel's MDS [Intel, 1975b] is just as complex and capable as many minicomputer systems; the CPU in the system just happens to be a microprocessor. The functions and facilities of a development system can be learned by the same methods we currently use for stand-alone, hands-on minicomputer installations used for teaching.

Applications

It is appropriate to conclude this section by again pointing out that it is the low cost of microprocessors and the resulting proliferation of applications that makes microcomputers different from minicomputers. The economic factors make it increasingly likely that a single engineer or a small group of engineers will be responsible for the complete design of a microcomputer system. This makes it essential that microcomputer engineers be familiar with both the software and hardware aspects of design and be able to carry an entire system design from initial specification to production release. Just as we cannot now support many programmers who only know how to write mathematical subroutine packages, we cannot have many logic designers who specialize in floating-point units or DMA modules -- these functions will be available on a single chip. Most engineers will be responsible for integrating LSI functional modules into a working system for a specific application. They will need to understand both hardware and software concepts; they will be concerned more with simplicity and modularity of system design than brute performance; and they will spend more time specifying and documenting a system, making it maintainable, than actually designing it. These are the most important factors that require microcomputer education to be any different from minicomputer education.

5. A MICROCOMPUTER LABORATORY COURSE

It is our firm belief at Stanford that the best introduction to microcomputers is the existing introductory computer engineering curriculum. Although our introductory courses are evolving to make greater use of microcomputer examples and specifics, the basic concepts taught in these courses remain the same. For the student who wishes to specialize in microcomputer systems we offer a laboratory course with the following 1-quarter prerequisites from the existing undergraduate curriculum:

1. Introduction to Computer Organization, Machine and Assembly Languages. Programming in a higher-level language is assumed as a prerequisite. In this course the student learns basic computer organization, assembly language for a particular minicomputer, basic I/O programming, and simple data structures. With this background, the student can start writing microcomputer programs in the first week of the lab course. Possible texts for the course are [Stone, 1972], [Stone and Siewiorek, 1975] and [Gear, 1973].

2. Digital Computer Organization. This course covers basic switching theory and logic design, logic families, computer arithmetic, processor organization, memory, input/output, and microprogramming. With this background, the student can easily grasp typical microcomputer system organization and he can easily design support circuits for a microcomputer system. Possible texts are [Mano, 1976] and [Gschwind and McCluskey, 1975].

3. Digital Logic Laboratory. In this course the student gains hands-on experience with TTL SSI and MSI integrated circuits. This course is necessary since each student in the microcomputer lab is given a set of microprocessor and support chips and required to build his own system. We don't want to spend time teaching the students things like which end of an IC is up. A possible text for the lab is [Wakerly, 1976a].

For some the prerequisites may seem imposing, but the first two courses should be required in any undergraduate electrical or computer engineering curriculum, and the third strongly recommended. So the prerequisites are a stumbling block only for students who are not electrical or computer engineers. It is obvious that increasing numbers of such students would like to learn about microcomputers so that they can use them in their own applications. But we believe that the background we require is necessary for anyone to become a proficient microcomputer designer.

The microcomputer laboratory at Stanford is a 3-unit, 1-quarter course. It includes approximately 15 hours of lectures and 40 hours of lab work. During the first five weeks of the course we give two lectures per week that cover designing and programming with a particular microprocessor. During the last five weeks we reduce to one lecture per week to survey other microprocessors. The first five weeks of lab work are specific assignments in which the student writes programs, runs them on a development system, and then builds his own hardware to run the same programs. In the last half of the course each student does a five-week final project of his own choosing. More details of the lab's format, equipment, and so on, can be found in another paper [Wakerly, 1976b].

6. RECOMMENDATIONS FOR MICROCOMPUTER LECTURE COURSES

There are a number of possible microcomputer lecture courses. We discuss several below. Possible texts to use in these courses are [Osborne, 1976a], [Osborne, 1976b], [Electronics, 1976b], and [Soucek, 1976].

Introduction to Microcomputers

We use this title to indicate a course for students with no logic design experience and no programming experience except for introductory programming with a higher-level language (HLL). A common misconception is that a one-quarter course on the assembly language of a specific microprocessor (say the 8080) constitutes a sufficient introduction to microcomputers. Although we've heard of such courses, we have found that it is impossible in one quarter to give students enough proficiency to actually design a microcomputer system. A solid introduction to microcomputers should be a 2-quarter sequence that includes the material of courses (1) and (2) in the previous section, using a microcomputer as the running example for both programming and system organization. In a compressed one-quarter course it might be possible to teach students some basic concepts and appreciation of computer capabilities, but they would be incapable of designing a system.

Introduction to Microcomputers for Logic Designers

For students with previous logic design experience, this course would cover the material of course (1) using a microcomputer as the example computer. The course could cover this material plus specifics on microcomputer hardware in one quarter. However, the

demand for such a course is likely to be low since most serious logic designers take course (1) as part of their curriculum.

Introduction to Microcomputers for Assembly Language Programmers

There should be plenty of demand for this type of course; it is much more common to see an assembly language programmer who knows little about logic design than vice versa. Students in this course would need only a few lectures to learn typical microcomputer software, from assembly language to HLLs to text editors and system programs. The bulk of the course material would be from course (2) of the previous section, using microcomputer examples.

Microcomputer Architecture and Applications

This would be an advanced course with courses (1) and (2) as prerequisites. The first several lectures could be devoted to the details of the hardware and software of a particular microprocessor, much like the lectures of the lab course described elsewhere [Wakerly, 1976b]. The remainder of the course would compare the important architectural features of different microprocessors and discuss important applications and advanced system architectures such as multimicroprocessors. The challenge in designing such a course would be to concentrate on the areas that are truly of general interest, avoiding entwinement with minor details of a particular microprocessor or system.

7. A COMMENT ON ASSEMBLY VS. HIGHER-LEVEL LANGUAGES

In the previous sections we have assumed that assembly language programming is a necessary skill for microcomputer system designers. To some this may seem unreasonable because of the growing trend to program in a higher-level language (HLL) whenever possible. Indeed, most large microcomputer software systems are being programmed in HLLs despite the relative crudeness of the existing microcomputer HLLs.

There are a number of arguments to support the requirement for assembly language programming. First, the existing HLLs do not make the microprocessor architecture transparent; hence familiarity with the machine architecture and assembly language is necessary. Even without this dependence, an HLL programmer may want to have a small portion of his system coded in assembler for efficiency reasons. The discussion in Section 4 on applications implies that there probably will not be separate departments for assembly and HLL programming; one programmer must be able to do both.

A second argument is that assembly language programmers tend to be familiar with input/output programming, interrupts, and real-time programming, whereas HLL programmers do not. Although this is caused as much by the emphasis given in teaching the two groups as by the level and applications of the languages, these programming areas are very important in most microcomputer applications. It may be that the teaching of HLLs in the future will evolve to cover these areas in more depth, or that future microprocessors will process HLLs directly. But for the moment assembly language is necessary.

A final argument is that a good assembly language programmer can quickly learn to write good programs in an HLL, whereas the reverse is not often true. By "good" of course we mean the modern notion of well-organized, documented, reliable, and maintainable programs rather than the old standard of fast, short, or tricky programs. Indeed, the only danger in teaching assembly language programming is failing to convey this notion of "goodness" and the fact that HLLs are suitable for 95% of the code in most applications.

8. CONCLUSION

The need to teach computer engineering students about microcomputers is undeniable. What is most important is to recognize that the impact of microcomputers is to make digital systems education more important in general. Microcomputer education does not require computer engineering departments to teach anything that is much different in principle from what they have been teaching for the past 10 years, although they may have to invest in new lab equipment to support that teaching. The biggest effect of microcomputers on education should be a general strengthening of digital systems education at the undergraduate level.

REFERENCES

DEC, 1973. PDP-11/05/10/35/40 Processor Handbook, Digital Equipment Corporation, Maynard, Mass.

DEC, 1975. LSI-11 PDP-11/03 Processor Handbook, Digital Equipment Corporation, Maynard, Mass.

Electronics, 1976a. Special Issue on Microprocessors, Electronics, vol. 49, no. 8, April 15, 1976.

Electronics, 1976b. Microprocessors, Electronics Book Series, McGraw-Hill, New York.

Fairchild, 1976. F8 Microprocessor Data Book, Fairchild Semiconductor, Mountain View, Calif.

Gear, C. W., 1973. Introduction to Computer Science, Science Research Associates, Chicago, Ill.

General Instruments, 1975. CP 1600 16-bit Microprocessor, General Instrument Corporation, Newport Beach, Calif.

Gschwind, H., and E. J. McCluskey, 1975. Design of Digital Computers, 2nd. Ed., Springer-Verlag, New York.

Hewlett-Packard, 1972. A Pocket Guide to the 2100 Computer, Hewlett-Packard, Palo Alto, Calif.

Intel, 1973. 8008 Users Manual, Intel Corporation, Santa Clara, Calif.

Intel, 1975a. 8080 Microcomputer System Users Manual, Intel Corporation, Santa Clara, Calif.

Intel, 1975b. MDS-800 Microcomputer Development System Operator's Manual, Intel Corp. Santa Clara, Calif.

Mano, M. Morris, 1976. Computer System Architecture, Prentice Hall, Englewood Cliffs, New Jersey.

Motorola, 1976. M6800 Systems Reference and Data Sheets, Motorola Semiconductor Products, Inc., Phoenix, AZ.

National, 1975. PACE Technical Description, National Semiconductor, Santa Clara, Calif.

Osborne, A., 1976a. An Introduction to Microcomputers (2 vols.), Osborne and Associates, Berkeley, Calif.

Osborne, A., 1976b. 8080 Programming for Logic Design, Osborne and Associates, Berkeley, Calif.

Signetics, 1975. 2650 Microprocessor, Signetics Corp., Mountain View, Calif.

Soucek, B., 1976. Microprocessors and Microcomputers, John Wiley, New York.

Stone, H. S., 1972. Introduction to Computer Organization and Data Structures, McGraw-Hill, New York.

Stone, H. S. and D. P. Siewiorek, 1975. Introduction to Computer Organization and Data Structures: PDP-11 Edition, McGraw-Hill, New York.

Texas Instruments, 1975. TMS 9900 Microprocessor Data Manual, Texas Instruments, Dallas, Texas.

Wakerly, J. F., 1976a. Logic Design Projects Using Standard Integrated Circuits, John Wiley, New York.

Wakerly, J. F., 1976b. "A Microprocessor Laboratory Course," Digest of Papers, DISE Workshop on Microprocessors and Education, Fort Collins, Colorado.

CHARACTERISTICS OF A REAL TIME COMPUTER SYSTEMS COURSE FOR ENGINEERS

R. A. Volz
T. L. Boardman
The University of Michigan
Ann Arbor, Michigan 48109

Abstract

This paper describes a new lecture-laboratory course being developed at The University of Michigan to provide students with technical knowledge and experience in incorporating computing elements into engineering designs. The course, intended for students with minimal computer background (e.g. one higher level language course), seeks to expose students to all aspects of real-time computing. These include: the significance of various computer architectures to real time operations; software considerations in real time applications; interrupts and device handling; analog to digital and digital to analog conversion techniques; computer to computer communication; and an introduction to digital process control. A laboratory, composed of three LSI-11 microcomputers systems and hardware necessary to connect these to external devices, provides students with hands-on experience in the application of these topics.

Introduction

The field of digital computers and their application is perhaps the most dynamic field in engineering at the present time. Driving this change during the past four years has been the introduction and widespread acceptance of the microcomputer. Already there are numerous products on the market using microcomputers, and the future is almost limitless. Recent articles in the technical [1, 2] and engineering education [3] literature provide strong evidence that this trend will continue. With the scope and magnitude of the use of microcomputers in the future, it is vital that engineering educators bring to their students the background necessary to implement systems with these computing elements as components. This paper describes a new lecture/laboratory course being developed at The University of Michigan to provide students with hands-on experience in the application of minicomputers and microprocessors as process control elements, data loggers or device controllers and communication front ends on other machines. The heart to this course is a set of new micro-computer equipment obtained with the assistance of the National Science Foundation*.

Often courses which focus on one aspect of computing become very specialized and can be taken only by students with substantial background in computing. In the case of real-time application, this situation cannot long prevail. The rapid drop in price and size of computing elements coupled with a very large rise in the performance/price ratio in the last few years has opened enormous application markets for microprocessors. During the past decade and a half we have seen computers come to be so heavily used for scientific computations that virtually all engineering students now take an elementary programming course early in their educational careers. Similarly, in the future, the broad range of application of microcomputers is certain to dictate that a sizable fraction of students for all engineering disciplines also acquire a background in real time application of computers. The new course described here is the first of several to be developed at Michigan and is intended to fill the need for an introduction to real time systems for a broad domain of engineering students.

The course is the result of a committee which was established to study the perceived need for College of Engineering support of Manufacturing Automation and Process Control. The committee, composed of faculty from throughout the College of Engineering, determined a specific lack of a course in Real-Time Computing Systems, not only within the College but within the University. It was recommended that such a course be directed to students with minimal programming and computer hardware background so that it could be taken by students with a primary interest in the application of Real-Time Computing Systems as opposed to interest in computer technology. For that reason it was decided that such a course should include extensive hands-on application of the concepts presented in the lectures. It was felt that such hands-on experience would require dedicated minicomputer systems suitable for Real-Time Programming and interfacing to external devices. Since a survey of the Engineering Departments indicated such minicomputer equipment was not available, the Committee submitted a matching funds equipment grant to The National Science Foundation. The dollar limit for such equipment grants of $40,000 was adequate for the purchase of three microcomputer work stations and a set of bipolar microcomputer hardware for individual experiments. The National Science Foundation funded the proposal starting July 1, 1975, which made it possible to purchase the equipment and develop a number of laboratory experiments so that the course could be first offered in Winter Term beginning January, 1976.

Equipment

Figure 1 shows the overall equipment configuration for the real-time systems laboratory. It consists of three LSI-11 microcomputer systems, each with a floppy disk drive, connection to our time-sharing system, Decwriter terminal, and various analog to digital and digital to analog converters. The microcomputers are configured so that they can be run independently to control analog and digital devices, each with its own access to our time-sharing systems for program development, cross compiling, and down loading. In addition, the computers can be connected together to simulate a small hierarchy computer system and allow the students to experiment with computer to computer techniques.

The LSI-11 microcomputers were selected primarily because of their 16-bit architecture and their compatibility with existing PDP-11 software. In addition, the bus architecture of these microcomputers makes them easily interfaced to external devices such as the floppy disks and analog conversion devices. The somewhat slower execution time of the LSI-11 microcomputers (approximately four microseconds per addition) is actually advantageous since it emphasizes processing delay times within real-time computer systems.

*Work described in this paper was, in part, funded by National Science Foundation Grant Number HES-7511412.

We decided to design and build the interface between the floppy disks drives and the LSI-11 computers ourselves and include the design procedure and construction as part of the course lectures and laboratory. The disks provide not only improved system flexibility in terms of local program storage and buffer areas, but also provide a very good example of a high speed real-time device which must be serviced meeting externally determined timing requirements.

We feel it is essential that each microcomputer station have independent access to our time-sharing system. To provide this access, each microcomputer is equipped with a serial interface and modem. The modems for two of the computers are standard 300 acoustic couplers. The third is a 1200 baud unit intended to provide communication to the time-sharing system for all three devices when the computers are connected hierarchically. This allows us to use the program development, editing, assembly, and high level language compilation facilities of the time-sharing system. We believe it is unreasonable to perform those operations in a machine as small as the LSI-11 microcomputers.

Probably the most important part of the real-time systems laboratory is the collection of external analog type devices which the real-time systems can control. In order to simulate virtually any real-time device, an analog computer for each microcomputer station was required. These analog computers, once interfaced to the digital to analog and analog to digital converters on the LSI-11 computers, allow us to simulate the operational characteristics of the real world devices allowing students to experiment with real-time operating systems and control algorithms.

In addition to these analog computers, we included $1100 for N-guage model railroad and interface equipment to be connected to one of the microcomputer stations. The model railroad configuration consists of six main lines which intersect at various points and an extensive humpyard siding for temporary storage of individual cars. The track layout is divided into sections which are electrically isolated so that multiple trains and switch engines can be runned simultaneously. Optical sensors are placed throughout the track layout to provide input to the microcomputers as to the current location, speed, and length of the various trains running.

Software

The code for most current real-time microcomputer applications is written in either assembly or machine language without the benefit of normal operating system functions. For the broad domain of students for whom the course is intended, this mode of operation is unacceptable. Too much time would be spent learning the details of assembly language programming. Moreover, with the broad range of application to be developed for microcomputers during the next decade, the conveniences and documentation support of high level languages will become increasingly important. Though one may use assembly language in high volume production situations, the greater programming efficiency of higher level languages, and decreasing cost of memory, and the high cost of maintaining assembly language programs will almost certainly make it more cost effective to do most programming in a higher level language which supports real-time operations.

To provide an environment in which students can readily develop real-time applications, a small operating system and a cross compiler (CRASH - Compiler for a Real Time Application SHop) are being developed. Since no one application is likely to need all of the facilities of the operating system (OS), the system could be modularized so that only the needed portions could be linked with user programs. However, for instructional purposes, the system will be left as a unit and always loaded. Its basic features are:

1. Input/Output (I/O) device drivers for the A/D's, D/A's, disk, paper tape reader, and a modem.
2. Protocol to allow the microcomputer system to be used as a terminal with the University's central computer.
3. An absolute loader.
4. Floppy disk file support.
5. A simple command structure modelled after the Michigan Terminal System (MTS) [4].
6. A small set of utility routines to support arithmetic conversion.
7. A wait structure to allow processing and I/O to proceed in parallel.
8. A task scheduler which functions with an external clock to start tasks at specified intervals subject to a specified software priority.

The system software configuration is structured to take advantage of a number of facilities available on the University's central computer facility. All assemblies, compilation, library searches and link editing of programs will be under MTS. The output of the link editor is in absolute loader format hence, the operating system need only have a small absolute loader. The modem will allow the user to treat the microcomputer as a terminal to MTS for programing preparation. When the absolute loader version of his program is ready, a protocol will allow him to either load his program into LSI memory directly from MTS, or transfer it to a file on the floppy disk. Either way avoids any need for the user to manually transfer tapes from one location to another.

The I/O and interrupt support structure is oriented towards real-time applications. All I/O requests do an immediate return to the calling program after the request is initiated so that processing may continue. Both record and character I/O are supported. When a user requires that an I/O task be completed before he can proceed, he must use a WAIT call. If a physical device is in use when an I/O request is made, an automatic WAIT is done. It will also be possible for a user task to be started upon completion of an I/O operation.

Each user load module defines one or more tasks which are known to and scheduled by the OS. A task may be:

1. run once, either immediately or at a specified instant in time;
2. run periodically every T_1 seconds;
3. run T_2 seconds after the completion of another task;
4. run at the occurance of some condition, e.g. interrupt from a specified device;
5. or run as a background task.

Timing is handled by an externally programmable real-time clock which has been built for the system. Each task has a software priority in addition to its scheduled timing; once stated, a task runs to completion or until interrupted by a task of higher priority.

The compiler is a cross compiler which will run under MTS and produce LSI-11 assembly language code which may in turn be run through an existing cross assembler. The CRASH languague is block structured and in some respects similar to PL/I, with the addition of constructs

to support real-time activities and the restriction of a number of unessential PL/I features, Only the unique features to the language will be discussed here.

The data types of the language include:

* ANALOG	* BOOLEAN
* BIT	* INTEGER
* CHARACTER	* REAL
* DISCRETE	* LABEL

INTEGER, CHARACTER and REAL variables may be declared as arrays with an arbitrary number of dimensions. Each dimension is specified as (Index_max:Index_min: increment).

The ANALOG variables are intended to be used with one of the analog to digital or digital to analog converters. The declaration of the variable must include a logical I/O unit through which the I/O is to be accomplished, a scale factor and offset to convert exernal voltages read to the engineering units used in the internal representation. If VALUE, SCALE and OFFSET are respectively the value produced by the analog to digital converter, the scale factor and the offset, the result produced is

$$\text{OFFSET} + \text{VALUE} * \text{SCALE} \qquad (1)$$

Each use of the ANALOG variable in a GET or PUT statement results in the appropriate I/O operation being applied to the specified device along with the offset and scale factor. No explicit statement of the logical unit number or calculation of the scaled value is necessary.

One common use of analog variables is in difference equations used to compute a new value for some output variable. Typically in such uses one needs past sample values of the analog variable as well as current values. Accordingly, a set of past values for each ANALOG variable will be automatically kept, and may be referenced with the forms

```
THETA@I
```

or

```
THETA@(EXP)
```

where THETA and I are declared ANALOG and INTEGER variables, respectively and EXP is an INTEGER, REAL or LOGICAL expression whose value is converted to INTEGER before use if necessary. Reference to past values is relative to the current value. The constant I and expression EXP are called delay indicators; that is, if I = 0, we are referring to the current value, if I = 1, the previous value, etc. The maximum depth of storage of past values must be specified with the declaration. This type of reference to a variable will be called a delayed reference, or reference to delayed values. Internally, the ANALOG variables may be stored as either INTEGER or REAL variables. The specification is made by using the attribute ANALOG in the declaration of the appropriate REAL or INTEGER variables.

In control applications, it is often desirable to manipulate a group of bits as a unit, e.g., the status word of a device. The DISCRETE type is defined to handle this situation. Each bit of a contiguous subset of bits of a DISCRETE variable may be given a name distinctfrom the DISCRETE variable of which they are a part, by using the DISCRETE attribute on an INTEGER declaration and in that declaration the names and position of bit variables. Reference to the bits of a DISCRETE variable may be by the name of the DISCRETE variable if they are to be manipulated as a group, or by the name of the component if they are to be treated separately.

The compiler will have the ability to optionally perform subscript range checking on arrays and delay depth checking on ANALOG variables. This checking may selectively be turned on or off individually, variable by variable, through use of compile time facilities. Overflow indicators may be associated with interrupt and special condition processing as described below if a variable is not assigned to an overflow indicator and an overflow occurs, a warning message will be printed and processing will continue.

A macro facility is also provided. A macro may encompass an arbitrary number of statements and nested macros up to five levels are allowed. For convenience, the compiler implementation will have a number of predefined macros (which may be redefined if desired). These will include mnemonics for default device assignments to logical units, common abbreviations, and common control constructs (such as DO FOREVER).

The usual arithmetic and logical operations and order of precedence hold. Mixed mode expressions are allowed with conversions performed as required.

In addition to the normal operations, certain of the arithmetic, logic and relational operators are extended to array operations. Addition (+), subtraction (-), AND (&) and OR (|) operations between arrays of matching dimensions are allowed and are executed element by element in row major order on the corresponding elements of the two arrays involved. In performing these operations, or any to be described subsequently, ANALOG variables are treated as one dimensional arrays of size specified by the depth of delay stored.

Multiplication between a scalar and an array, and division of an array on the right by a scalar are defined and are executed element by element in row major order by applying the operator between the elements of the array and the scalar. Multiplication between one and two dimensional arrays of appropriately matching dimensions is also defined.

The language provides a variety of control structured programming. It has DO WHILE and DO with iteration count constructs as in PL [5]. Though redundant and not essential, a DO UNTIL construct is included for convenience and ease of reading programs. In addition, a DO CASE construct as in XPL [6] is provided. Two forms of IF statement are provided: IF...THEN...; and IF... THEN...ELSE.... These function as in XPL.

To provide for program structure and modularity, procedures may be defined and called. Procedure definitions may be external and separately compiled or internal and placed anywhere within the program. Procedures will be nonrecursive with calls by reference, except for constants which will be by value. The usual variable scope is in PL/I or XPL will apply. Though its use is not encouraged, a GO TO statement is provided to avoid certain awkward situations.

There are two basic types of I/O statements. The first is an ALGOL-like INPUT-OUTPUT pair which will primarily be used for communication with the user's console. The second class of I/O is through GET and PUT statements. These are intended primarily for use with a variety of external devices. With a PUT or GET, the logical device number (LDN) is attached to the variable at declaration time rather then to the I/O statement itself. A built-in function allows reassignment of the LDN.

Course Outline and Projects

As mentioned above, this course is intended to provide a person with minimal programming and hardware background knowledge about and experience with real time computer systems. The course covers the characteristics of real time operating systems, documentation guidelines for those systems, basic digital logic including linear and some nonlinear integrated circuit descriptions, mini and microcomputer hardware organization, analog to digital and digital to analog conversion techniques and devices, multiplexing, digital and analog communication characteristics, basic control theory, computer to computer communication techniques, and networking.

A real-time operating system which provides an interface between user programs and external devices such as teletypes, disk units, A/D and D/A converters, and the time sharing systems; and provides multi-task processing, external interrupt handling, and a real time clock, is available for student use throughout the course. This real-time operating system is described in detail in lecture along with the rational for its specific characteristics and capabilities. A complete user oriented document on the system is also available to the students and is used as an example of good documentation procedures.

It is felt that an essential portion of real-time systems use involves physically interfacing such systems to existing real world devices. For that reason, a portion of the course is dedicated to basic digital logic, in particular, series 7400 TTL digital logic. An introduction to these 7400 series devices is presented and the design of the disk interface which connects the floppy disk units to the LSI11 microcomputers is detailed. The students are provided with the resulting logic diagram of the interface which they can compare to the actual circuit boards connected to the LSI-11 machines.

Attention is also paid to the architecture of the LSI-11, in particular with respect to its external interfacing characteristics. In addition, other minicomputer systems are discussed and their architectural differences and similarities highlighted.

Since most external devices involve both analog input and output, a portion of the course is dedicated to analog to digital and digital to analog conversion techniques and characteristics of commercially available A/D and D/A converters. Most laboratory experiments throughout the course involves using A/D and D/A converters which are part of the overall microcomputer stations. The design for the interface between the LSI-11 microcomputers and these converters, which was developed by independent study students, is also provided to the students.

A number of laboratory projects use the model railroad layout. First, simple analog and digital control tasks are studied by driving train speed in each track section and switch position. Next, external event sequencing is studied by student projects which set specific train routes. Finally, the students implement queueing algorithms for the assembly and disassembly of trains to and from the humpyard siding. Naturally, the real-time systems and control algorithms implemented on them must be capable of accepting train location interrupts on a continuous basis and use that location information to prevent collisions. We believe control of such a model railroad, in addition to providing interest and excitement in the laboratory, is an exceptionally good demonstration of many of the real time control problems.

Since this course is oriented toward the control of real world devices with computer systems, a portion of it is devoted to various control algorithms and control system characteristics. Open and closed loop systems performance measures techniques for determining control systems stability and simple controller types are mentioned. The students are required to perform a laboratory experiment involving the closed loop control of a weighted fly-wheel system.

Finally, operating system requirements for computer to computer communication are covered, as well as the hardware characteristics of such communications systems. We feel this is necessary for systems involving hierarchies of computers.

Summary

The material being developed for the new course fills an important need at Michigan. The developments described are too recent for proper evaluation at this time, however, there will be considerable further development over the next few years. Nevertheless, student acceptance has been high and the demand for the course far exceeds our resources for offering it.

Acknowledgements

There are a number of students who have contributed significantly to the development of the real-time computer laboratory. Ted Kowalski and Jack Bonn have been the principals in the development of the real time operating system for the LSI's. The compiler was designed and written by a project class consisting of Jack Bonn, Brian Cashman, Ted Kowalski, Alex Kushmer, Steve Medlin, Bert Moberg, John Puttress, Al Segal, Jim Sterken, and Dave Yeager. Steve Medlin and Brian Cashman have continued the work on the compiler since the end of the class. Much of the hardware interface design and construction has been done by Charles Rothchild through an independent study project and Bill Dargel during a summer project.

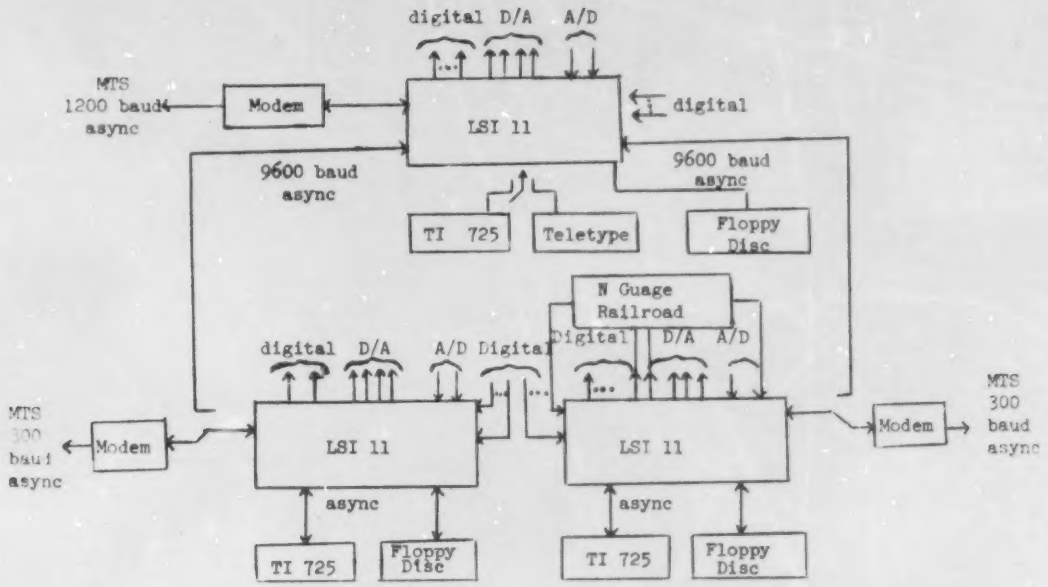


Figure 2. Organization of Microprocessor Equipment

References

1. C. G. Bell and Allen Newell, Computer Structure: Readings and Examples, McGraw-Hill, New York, 1971, Chapter 1.

2. Robert A. Ellis, "Modular Computer Systems" Computer, Vol. 6, No. 10, (Oct. 1973), pp. 16.

3. C. G. Bell, John Grason and Allen Newell, Designing Computer and Digital Systems of Using PDP-16 Register Transfer Modules, Digital Press, 1972.

4. University of Michigan Computer Center, Vol. 1, MTS and the Computing Center, January 1973.

5. IBM, PL/I Language Reference Manual, copyright, IBM, Dec. 1972.

6. McKeeman, Horning & Wortman, A Compiler Generator, Prentice-Hall, 1970.

TWO DIVERSE APPROACHES TO THE DEPLOYMENT OF MICROPROCESSORS IN A COMPUTER ENGINEERING CURRICULUM

G. R. Kane, H. T. Nagle, Jr., L. E. Hand
Electrical Engineering Department
Auburn University
Auburn, Alabama

G. L. West
Texas Instruments
Dallas, Texas

Summary

The need to introduce microcomputer chips and systems is causing Auburn University's Electrical Engineering Department to establish two courses directly relating to microprocessors but treating them from different points of view. These differing approaches support the contention that microprocessors offer no radically new subject to computer engineering but provide a feasible way for dramatically increasing student interest and progress.

Introduction

The continued rapid development of microcomputer chips and systems requires that these devices be utilized in Computer Engineering curriculums as soon as practical. Auburn University is using microprocessors as an integral part of two courses--the first is an advanced senior-graduate level elective aimed primarily at microprocessor structure and application and the second which is a required course within the Electrical Engineering Department.

Course Structure in Auburn's Computer Engineering Program

At Auburn University the computer is a prime tool in the Electrical Engineering program. Four required courses have a direct connection to computer use, organization or design. Many others employ the computer as a design tool such as computer-aided integrated circuit analysis. The required sequence consists of EE 201 - Introduction to Electrical Engineering (with emphasis on programming in Basic and Fortran), EE 322 - EE 324 a two course sequence on combinational and sequential circuit design, and EE 425 - Minicomputer Organization. The upper division courses form a 10 quarter-hour block which is the base of a Computer Engineering option which includes advanced undergraduate electives in Systems Programming, Compilers, Artificial Intelligence, Fault Diagnosis, Computer Graphics, and Microprocessors. The Computer Engineering curriculum structure is shown in Figure 1 [1]. One of the few redeeming qualities of a quarter system is its ability to support a large number of diverse courses that would have to be combined in a semester system.

Computer Engineering is a discipline that not only requires adequate foundation in hardware concepts but also clear regard for software solutions as well. Perhaps no other subsection of computer engineering reflects this idea more than microprocessor systems. If the microprocessor were simply another LSI component its application growth would not have been as spectacular. The true power of these little devices is in their flexibility, their ability to be programmed.

We now describe in moderate detail two courses in the computer engineering curriculum of Figure 1 which employ microcomputer systems: EE 425, Minicomputer Organization, and EE 524, Microcomputers.

Minicomputer Organization

EE 425, the last required course in Auburn's computer engineering base, has been used to teach computer organization and low level programming. The 4 quarter-hour course consists of both lecture and laboratory. Under an NSF grant the course is being reconstructed to include the microprocessor and provide student/machine interaction in a way not possible before the advent of microcomputers.

The minicomputer made possible a revolution in computer science by allowing students greater contact with the hardware than was possible with the large main frames. As the minicomputers matured, the hands-on feature fell into disuse. One early machine, Washington University's LINC, had exceptional console features that included hardware single-step, trace, breakpoint along with the ability to display the entire machine state. As the complexity of these small systems grew, interactive features were discarded until present consoles are rather austere. This is chiefly a matter of economics. The costs of peripherals are not decreasing at the rate experienced by the miniprocessor. A small system consisting of a minicomputer and a terminal can often be economically operated by one user. The sophistication of the application increases the demand for backing store and fast hard copy. These expensive items cannot be justified for a single user. This single user is not taxing the minicomputer systems, so larger operating systems are installed to permit a variety of users to have access to the machine. In this configuration the presence of a sophisticated console is of little importance as hardware breakpoint and single step features cannot be exploited without catastrophic interference to the rest of the system. When attempting to employ these more sophisticated systems, fundamental principles in the design and utilization of the computing machine are often obscured. Consider the concept of an external interrupt. This feature is so necessary to the development of computer systems that it is difficult to imagine systems operating without them. In a complex system the management of interrupts is so critical that it is not left to the casual user. An inexperienced individual (student) can gain expertise in this subject only after being experienced. The circular nature of the previous statement is symtomatic of the problem at issue here--the microcomputer offers the opportunity to break this circle by proving genuine machine contact. We are presently developing ultra-portable minicomputers based on present microprocessor circuits in sufficient quantity to permit individual use.

Outline. The following objectives are sought in employing personal computers.

1. The number of machine related projects (programs) is increased from the four projects normally assigned when sharing a machine, to fifteen per quarter. Individual machines mean that programming projects are as easy to assign and solve as homework.

2. Certain fundamental topics can be covered that are not possible in a batch environment (e.g. interrupts).

3. Related topics such as interface design for these small machines will be studied.

EE 425 - Minicomputer organization includes the topics: Hardware organization of a simple computer (primary memory, control, arithmetic, data representation, instruction cycle, addressing modes); assembly language programming (program flow, subroutines, macros, scaling, Input/Output processing); and hardware design (microprogramming, data buses, Input/Output control and interfacing). These are supplemented by the laboratory/homework projects. A sampling of which includes

1. Basic Machine Organization and Data Flow
2. Straight line Programs
3. Addressing Modes
4. Program Loops
5. Program Tracing
6. Subroutine Mechanisms
7. Input/Output Techniques
8. Interrupt Mechanisms - Program Traps
9. Interrupt Mechanisms - External Events
10. Interface Design

Facilities. The Motorola M6800 microprocessor was chosen for its instruction set which we feel to be superior for educational purposes.

The machines consist of a front panel with a BCD as well as binary display and a switch register, 512-to-1K bytes of read/write memory and a stored monitor in ROM (256-512 bytes). It is not possible to display the contents of all machine registers at all times; the system limitations of using an existing microprocessor chip with pinout already specified dictate the use of a simple monitor program to provide register and memory display and modification as well as the single step and slow mode features mentioned earlier.

External interrupts are to be provided by a line frequency derived clock. An additional uncommitted I/O port is included to permit simple interface design. Several such interfaces suggest themselves immediately. A pair of inexpensive D/A converters, and an oscilloscope will provide a very modest graphics processor, a D/A converter and a comparator can be used to form a low speed A/D system. Clearly, this list can be expended almost without limit.

EE 425 under the reconstructed format employing individual computers will be taught for the first time during the Winter Quarter 1977.

Microcomputers

Once a student has learned the basic concepts of minicomputer organization at the register-transfer level (EE 425) and has had an introduction to logic design (EE 322), he is ready to concentrate his studies on microcomputers and microprocessors (EE 524).

Outline. The course begins with a survey of microcomputer systems. Then the INTEL 8080 and Motorola 6800 processors are examined in detail at the instruction set level. Next the microcomputer software development cycle and programming techniques are studied. The hardware for interfacing to the 8080 and 6800 systems is examined and the course concludes with a survey of microcomputer applications. A hardware/software laboratory accompanies the course.

Facilities. EE 524 is supported by a microcomputer laboratory with equipment ranging from chip sets to a timeshare system. See Figure 2. A detailed description of these facilities appears in [1]. For software production, cross assemblers in BASIC are available on the RSTS/E system to generate object files from source files. A student enters his program via the card reader into a source disc file, or he may use the RSTS/E edit features to prepare his source code directly on disc. The object file is sent via a RSTS/E multiplexer port directly to the target microcomputer's memory. Once the object code has been loaded into the microcomputer's random access memory, execution may occur, or a programmable read only memory copy may be generated. Hence a student can prepare a source program, translate it to object code, and "burn" it in PROM without touching a paper tape! This system works well as a software laboratory tool.

Hardware experience is gained through experiments, such as bus loading, clock generation, input/output, and interrupts. Each student also participates in a class project designed to match a microprocessor to some application. Chip sets and breadboarding equipment are available for project construction and testing.

History. EE 524 evolved from a digital systems design course which surveyed modular logic design and the INTEL 8008 microcomputer. The course was offered twice per year and averaged 20 students per class. EE 524 in its present form will be offered for the first time in the Fall Quarter of 1976.

Conclusion

Two courses employing microprocessor hardware have been described. The first is a programming/machine organization course; the second, primarily a hardware design, applications course. The two courses reflect our belief that microcomputers are not just another new component in the digital scene. The situation is more like the familiar children's story with Every-So-Much-More-So. No matter how the stuff is applied it makes things better--either saltier, or sweeter, or slippery--just Ever-So-Much-More-So. The microprocessor is the Ever-So-Much-More-So for the computer engineer. The topics do not change but are rather repeated in a simpler more accessible fashion.

The tremendous cost reductions in microcomputer hardware over the last few years has made it possible for computer engineering educators to literally put the computer in the hands of the student. A concentrated effort in this direction is taking place at Auburn University and will continue in the future, perhaps even into our beginning logic course, EE 322. A modest investment in digital hardware can now yield many returns in student interest and progress.

Reference

[1] B. D. Carroll, H. T. Nagle, Jr., G. R. Kane, and J. D. Irwin, "Computer Engineering at Auburn University," 14th Annual Southeast Regional ACM Conference, Birmingham, Alabama, April 22-24, 1976.

Partial support provided by NSF Grant SER76-14870.

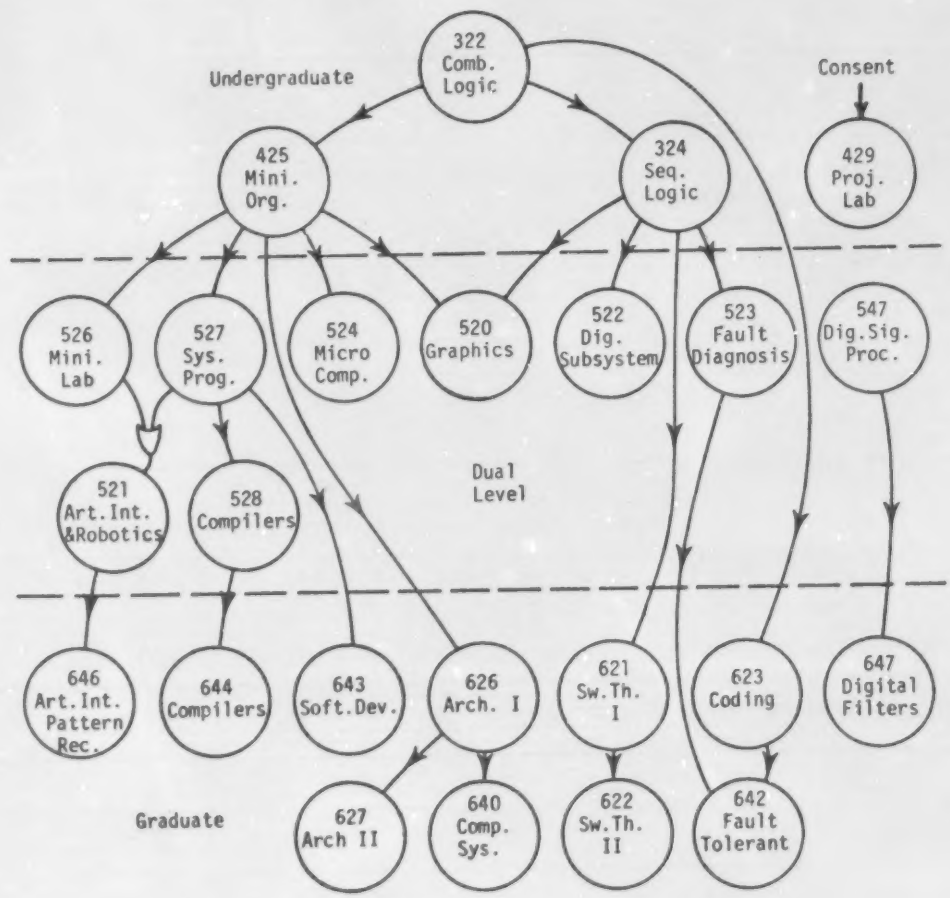


Figure 1 Prerequisite Structure

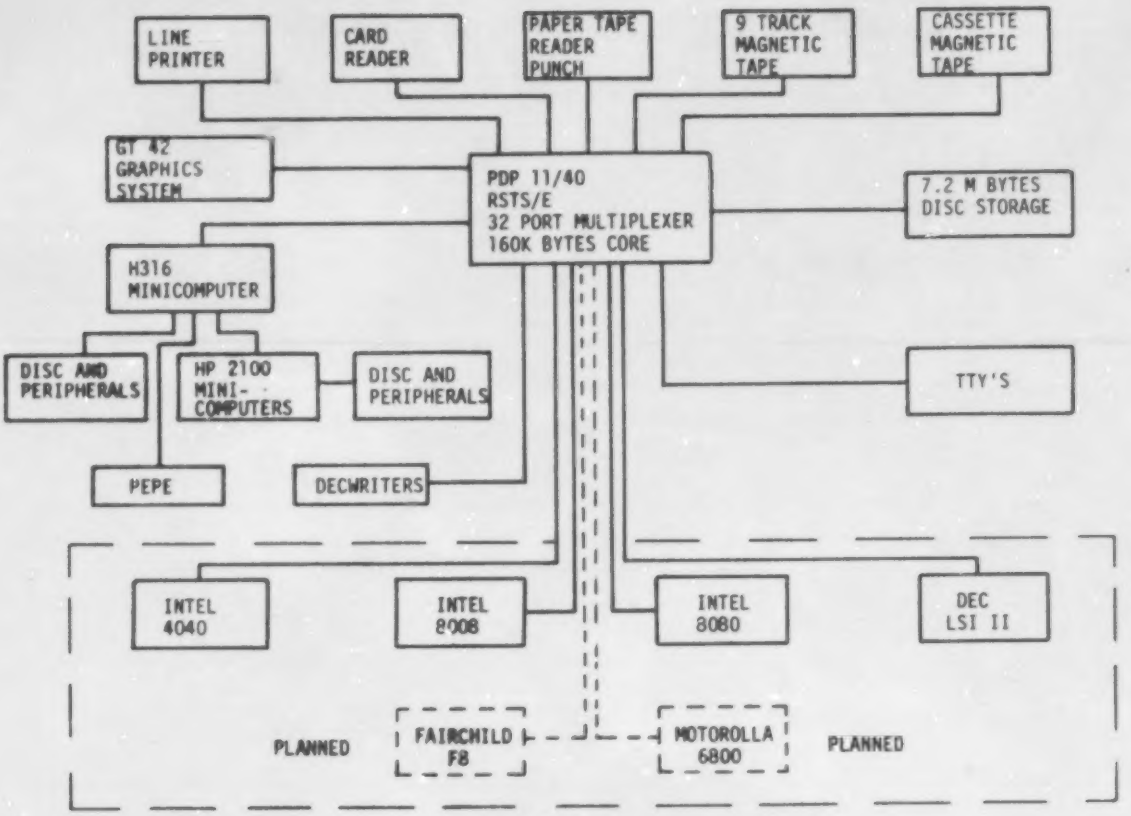


Figure 2. Computer Engineering Facilities

EDUCATION IN MICROPROCESSORS AND MICROCOMPUTERS

Lance A. Leventhal, Ph.D.
Engineering Technology Department
Grossmont College
El Cajon, California 92020

Summary. The new subject of microprocessors and microcomputers will soon be an integral part of most engineering curricula. This article deals with some of the difficulties involved in introducing microprocessors into the curriculum and describes a proposed series of courses. The importance of describing both the hardware and software aspects of microprocessors adequately is emphasized.

Microprocessors and microcomputers are obviously an important new topic in modern engineering. Short courses, conferences, and seminars on the subject have become commonplace; a large demand now exists for people trained in the microprocessor area. The number of papers and articles dealing with microprocessors and microcomputers has increased more than tenfold in the last few years. Clearly most engineering schools, regardless of level, will want to introduce microprocessors into their regular curricula. This paper offers guidelines for that introduction and describes a series of courses that have been presented or proposed at Grossmont College.

Microprocessors represent a breakthrough in the cost and minimum purchasable unit of computing power. Microprocessors do not perform any functions that cannot be performed by larger computers; in fact, standard microprocessors like the Intel 8080 or Motorola 6800 are far less powerful than even the smallest minicomputer. However, the low cost and small size of microprocessors has opened application areas in which the use of computing power could not formerly be justified. Instruments, terminals, small business machines, test equipment, data acquisition systems, computer peripherals, appliances, and automobiles are just a few of the many application areas for microprocessors.

The widespread use of microprocessors means that more engineers and technicians must become familiar with computers and more programmers must become familiar with the types of engineering tasks for which programs will now be written. Neither hardware nor software knowledge alone will be sufficient to work with microprocessors. Nor can the specialist in hardware or software be as isolated from the other specialty as is often the case with larger computers. This new union of hardware and software will present even more difficulties to the educator than has the similar situation in minicomputers.

Instructors and students with a hardware background will have to recognize that microprocessors are central processing units, just like those of large computers. Microprocessors may look like other LSI packages but they perform the same functions as any CPU. Microprocessors are computers, not variable operational amplifiers, logic state sequencers, or "smart wires." The computer languages, operating systems, and software and hardware methods that have resulted from twenty-five years of work with larger computers are applicable to microprocessors. In particular, such programming methods as modular programming, structured programming, top-down design, flowcharting and decision tables can provide a framework for developing software for microprocessors much as they have for larger computers. Microprocessor courses must introduce those with hardware backgrounds to modern software. Although most microprocessor programming currently is done in assembly language, the added convenience, clarity, and productivity of higher-order languages will eventually result in their widespread use with microprocessors much as with minicomputers. New languages may have to be developed for microprocessors but the principles of software design will be the same principles that have been derived from experience with larger computers.

However, instructors and students with software backgrounds must recognize that microprocessors are LSI devices which are used as an integral part of electronic systems. A microprocessor may be functionally the same as a large CPU, but is is physically the same as any other complex semiconductor device. Interfacing the microprocessor to a system will depend on physical and electrical characteristics that are dependent on the semiconductor technology from which the processor is constructed. Interfacing will also require the use of other integrated circuitry. The interface will have, in turn, a critical effect on the software. No microprocessor user is likely to be successful without a reasonable understanding of the hardware aspects of microprocessor-based systems.

Furthermore, microprocessors are largely used to solve engineering problems rather than to perform traditional computer tasks. Microprocessors are not used to handle complex scientific calculations or to process large numbers of business records. Instead, microprocessors are mainly used as controllers, calculators, and dedicated processors in such tasks as controlling displays, motors, or other mechanical and electrical devices, monitoring inputs from sensors and actuators, converting and formatting data and performing simple calculations. Most programmers, even those with real-time experience, will find that such problems require more detailed understanding of system hardware than do most tasks performed with larger computers.

Thus courses in microprocessors will require a combined hardware and software approach. Although most microprocessor courses will probably be given in electrical engineering departments, computer science departments will also have to become involved. Neither a hardware nor software approach alone will be sufficient to either fully explain microprocessors or to meet the needs of students with varied backgrounds who will have to use microprocessors. Both engineers and programmers will be involved in this new technology and each must have an adequate understanding of modern developments in the other's area.

Some preliminary guidelines can be defined for both the hardware and software sections of a microprocessor course. In the hardware area,

1. Basic digital integrated circuits must be stressed, with examples as to how SSI and MSI devices can be used with processors.

Many interested students and engineers cannot be presumed to have extensive background in digital integrated circuits. The Bugbooks (References 3, 4, and 5) include both experiments and discussions at a satisfactory level.

2. The use of standard input/output devices widely employed with microprocessors should be explained. Standard I/O devices include displays, coded and unencoded keyboards, A/D and D/A converters, teletypewriters, and paper tape readers.

3. The use of other LSI and MSI devices with microprocessors should be discussed. LSI devices include UARTs, serial and parallel interfaces, multipliers, interrupt and DMA controllers, calculators, and special purpose interfaces. MSI devices include counters, decoders, multiplexers, selectors, shift registers, and parity generators.

4. General design principles must be emphasized, including the basic functions of clocked systems, synchronous and asynchronous input/output and signal buffering and conditioning. Although some of the limitations of current microprocessors will soon only be of historical interest, the principles of system design will remain the same.

5. Semiconductor memory systems must be described as an intrinsic part of microcomputer technology. Such memories will continue to be a major cost item and design consideration in microcomputer systems.

In the software area, the following guidelines are important:

1. The entire software design process should be stressed--not just the programming aspects. Software design includes problem formulation, program design, debugging, testing, documentation, and maintenance and re-design as well as the coding of instruction sequences.

2. Proper programming technique, including clear program structure and thorough documentation, must be taught. This guideline also means that the instructor's examples must be well-formulated. The book by Kernighan and Plauger (Reference 6), although aimed at higher-order languages, offers many useful hints and examples.

3. Modern programming techniques should be described. Top-down design, structured programming, modular programming, and other similar methods not only have been shown to be productive but can offer a definite framework on which to base programming efforts. Such programming techniques are described in References 7 through 10.

4. Programming examples should be drawn from actual microprocessor problems. Typical programming examples like calculating factorials, solving quadratic equations, and summing integer series are really not relevant to the use of microprocessors. More meaningful examples would include code conversions (ASCII, EBCDIC, BCD, seven-segment), parity and other error checking procedures, editing functions, teletypewriter interface programs, self-checking numbers, formatting of data, linearization, and BCD and floating point arithmetic.

The problem of equipment has become more tractable with the introduction of a variety of inexpensive kits. These are available both from the semiconductor manufacturers and from other sources; many are described in Reference 11. Schools with even moderate laboratory facilities will find unassembled kits to be cheap and easy to put together; such kits should include a printed circuit board and a ROM-based monitor to be really useful. Assembled kits are also available. Many of the kits, however, require a teletypewriter which must either be purchased or rented. Another option involves kits that incorporate calculator-like keyboards and displays; such kits seem particularly attractive in the educational environment where large programs are seldom written anyway.

The microprocessors, even if purchased as assembled kits, will not tend to be the largest equipment expense. Memories, input/output boards, and various SSI and MSI functions can easily cost far more than the CPUs; many of the so-called bargains in integrated circuits only apply to large users. Software is a further problem. Few kits offer much more than a monitor/debugger package. Self or cross-assemblers usually cost several hundred dollars each and are only useful for a single processor. Simulator programs (which perform a software simulation of the microprocessor) are comparably priced. The few higher-order languages that are available are priced even higher (usually over a thousand dollars) and can run only on large computers. Clearly the companies which develop this software must recoup their large investment. However, most schools which attempt to offer microprocessor courses will find software to be a rather large stumbling block. Some arrangement may eventually have to be worked out between the educational community and the manufacturers; professional societies like the IEEE would appear to be obvious intermediaries.

A Sample Curriculum

Grossmont College currently offers or plans to offer the following microprocessor courses:

1. Introduction to Microprocessors

This course serves as a general introduction to microprocessor hardware and software. Topics covered include the comparison of microprocessors to minicomputers and other LSI devices, microprocessor architecture, instruction sets, assemblers, assembly language programming, software design and development, timing and control, memory systems, input/output, and interrupts. The Intel 8080 is emphasized as a typical standard microprocessor.

The associated laboratory covers basic integrated circuits, I/O devices, and the use of the processor and a minicomputer-based cross-assembler. Simple projects include the development of digital timers, digital thermometers, and digital scales.

2. Advanced Microprocessors

This course deals with the programming of microprocessors in higher-order languages and the use of microprocessors in communications and data acquisition systems. The use of peripheral interface chips, UARTS, chip modems, and other special-purpose devices is emphasized. The Motorola 6800 and MOS Technology 6502 are introduced.

3. Microprocessors and Microprogramming

This course deals with the use of bit-sliced bipolar microprocessors in microprogrammed systems. Topics covered include microprogram design and development, pipelining, look-ahead carry, ROM simulators, and microassemblers. The Intel 3000 is used as an example.

Tentative course plans or proposals include:

4. Calculators and Microprocessors

This course would cover the use of calculator chips

and simple microprocessors like the Intel 4040 or Rockwell PPS-4 as replacements for random logic.

5. Distributed Microprocessor Systems

This course would deal with the use of multi-microprocessors in parallel or distributed systems.

6. Hobby Computers

This course would describe the programming and interfacing of home computer systems.

Course one is now being offered for the second time while Course two is being offered for the first time. Course three will probably be offered in the Fall of 1976.

Particular processors to be used in these courses were selected on the basis of widespread use in engineering applications, availability in convenient form at reasonable prices, software support, and documentation. Surveys have shown relatively little difference in performance between microprocessors of the same semiconductor technology. The need for supporting hardware and software would generally require most courses to limit themselves to a particular processor even though the CPUs are relatively cheap.

This article has described some of the problems which must be faced if microprocessors are to be effectively incorporated into an engineering curriculum. A combined hardware and software approach has been emphasized as the best way to explain microprocessors. A proposed series of courses has been described which deals with a broad spectrum of processors.

Acknowledgements

Mr. William Tester of Grossmont College is responsible for the introduction of the courses described in this article into the college curriculum. Support and suggestions were also received from Mr. Colin Walsh of Grossmont College, Mr. Jeffrey Haight of Linkabit Corporation, Professors Fredric Harris and Nicholas Panos of San Diego State University, Mr. David Bulman of Pragmatics, and Dr. James Tiernan of the Naval Electronics Laboratory Center. Dr. Bernard Carey of the University of Connecticut provided an interesting viewpoint and copies of his unpublished work.

References

1. Ward, A. R., "LSI Microprocessors and Microcomputers: A Bibliography Continued," Computer, Vol. 9, No. 1, January, 1975, pp. 42-53.

2. Ward, A. R., "LSI Microprocessors and Microcomputers: A Bibliography." Computer, Vol. 8, No. 7, July, 1974, pp. 82-86.

3. Rony, P. R., and D. G. Larsen, "The Bugbook I," E and L Instruments Inc., Derby, Conn., 1974.

4. Rony, P. R., and D. G. Larsen, "The Bugbook II," E and L Instruments Inc., Derby, Conn., 1974.

5. Rony, P. R., D. G. Larsen, and J. Titus, "The Bugbook III," E and L Instruments Inc., Derby, Conn., 1975.

6. Kernighan, B. W. and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, New York, New York, 1974.

7. McGowan, C. L. and J. R. Kelly, Top-Down Structured Programming Techniques, Petrocelli-Charter, New York, New York, 1975.

8. Knuth, D., "Structured Programming with GO TO Statements," Computing Surveys, Vol. 6, No. 4, Dec., 1974, pp. 261-301.

9. Basili, V. R. and T. Baker, "Structured Programming: A Tutorial," IEEE Computer Society, New York, New York, 1975.

10. Miller, E. F. Jr., and G. E. Lindamood, "Structured Programming: Top-Down Approach," Datamation, Vol. 19, No. 12, December, 1973, pp. 55-57.

11. Torrero, E. A., "Just Getting Into Microprocessors?" Electronic Design, Vol. 24, No. 1, Jan. 5, 1976, pp. 46-50.

12. Sloan, M. E., "Potential Trends in Microcomputer Education," IEEE Compcon 74 Proceedings, pp. 131-134.

13. Booth, T. L. and B. J. Carey, "Impact of Micro/Minicomputers on Electrical Engineering Curriculums," IEEE Compcon 74 Proceedings, pp. 135-137.

14. Carey, B. J., "Standard Microcomputer Modules," University of Connecticut School of Engineering, 1975.

A MICROPROCESSOR LABORATORY FOR A UNIVERSITY ENVIRONMENT

Bernard J. Carey
University of Connecticut
Department of Electrical Engineering
and Computer Science
Storrs, Ct. 06268

Abstract

A microprocessor based laboratory facility is being developed as part of the Computer Science program in the Department of Electrical Engineering and Computer Science of the University of Connecticut. This laboratory is structured to support graduate research in which the microprocessor will play a role, provide a means for introducing microprocessor based material into the undergraduate computer science and electrical engineering curriculums, and be a medium whereby the untrained computer user who wishes to utilize a microcomputer may do so in a convenient manner. The laboratory is centered on a dual PDP 11 minicomputer system which provides a high level of software and hardware support to the microprocessor activity. This permits software development to take place in a comfortable, real time environment on a machine suited to such a task plus providing hardware support not inherent in the microcomputer itself. Since a broad spectrum of people will use this facility, the involvement with the microprocessors and related devices must cover a broad spectrum of activities. This will range from new designs starting at the device level to applications which simply integrate complex off-the-shelf hardware and software functions. To support this the laboratory provides a means to reliably breadboard a prototype of a new microcomputer system or sub-system. This would be done by an individual who is conversant with the intimate details of a given microprocessor device line and is knowledgeable in both hardware and software aspects of digital systems. For the person who is not conversant with or possibly not interested in the hardware device details, a variety of standardized, functional microcomputer modules are available to be integrated into a microcomputer system. The key feature of these modules is a standard input and output structure which imposes a circuit and functional compatibility between a wide variety of modules. This permits standard semiconductor memories, peripheral mass storage devices, computer terminals, analog input and output, and other such functional modules to be compatible with various microprocessors. This will also provide a basis for the present systems to be upward compatible with future technical advances. At present, the laboratory provides a means for working with 4 bit systems, 8 bit systems, and microprogramable systems.

Introduction

This article will discuss certain attributes of a laboratory facility organized around microprocessors, and related devices, for the primary purposes of supporting graduate research in Computer Science and for introducing microcomputer related educational material into the undergraduate curriculum of the Department of Electrical Engineering and Computer Science of the University of Connecticut. In addition, it will provide a service function to other scientific areas in the university community. Since this facility will provide a resource which is intended to be used in a university environment, there are certain technical, economic, and organizational constraints which are peculiar to such a situation. Possibly the primary characteristic of a microprocessor laboratory at a university is that it will be (or should be) a multi-use facility. By this it is meant that not only must the same physical resource service a variety of operations such as faculty research, undergraduate education, and the design and development of microcomputer based solutions to engineering problems, but it must also service a multitude of users whose expertise will range from the well trained computer scientist to the untrained individual who views the microcomputer as a black box which does "something" useful for him.

From an information processing viewpoint, there is nothing fundamentally new about the microprocessor and its associated devices. What is new is the cost and size advantages associated with the microcomputer which makes it economically and physically feasible to consider using digital processing methods in achieving solutions to problems which were traditionally approached with analog technology or, even more significantly, in evolving solutions to problems for which only abstract solutions previously existed. A transition from analog to digital techniques in control systems [1] may be an example of the first situation and the possible evolving of large multiprocessors is an example of the latter. Fundamental concepts in multiprocessors have been in existence for over twenty years. Early systems such as the TRW-400 [2] illustrated the basic concepts which are still used today but were physically unwieldy. Perhaps even more exciting than finally realizing solutions to standing problems in computer science are the possibilites of introducing miniaturized digital computers as tools which may be used in other scientific areas. A fruitful area is in biomedical engineering in which a miniaturized computational function may provide a variety of vital human services. A recently cited example was a proposed system for prothesis control [3] where a microcomputer was the interface between the human nervous system and an artificial limb.

This discussion is only intended to illustrate the variety of areas in which a microcomputer might serve a useful service. As is consistent with the overall history of digital computers, there is an inherent advantage, in an open ended number of applications, to using digital computation techniques. As the size and cost of the digital computer decreases the number of users and application areas increases. The microprocessor may be the logical end point of this process whereby the size and cost of the computing function are approaching zero (relatively) so that the number of users and application areas are becoming really large. Surveys [4,5] of literature relative to microprocessors seem to support this view. How may a university best cope with this situation? A university must not only educate its students to be able to effectively understand the microcomputer for use in their future work, but it must also provide a means whereby its personnel (student, faculty, staff) can utilize a microprocessor, a microcomputer, microcomputers, and any necessary peripheral devices in the broad spectrum of activities occurring in a modern university. Therefore, considering the variance in application areas and the type of user, the establishment of a general purpose, multi-use facility seems to be a cost effective approach to realizing a microprocessor laboratory in a university

environment.

The Impact on the Academic Curriculum

A topic of ongoing concern in various curriculums is how best to incorporate advances associated with microprocessors into the educational process. The most immediate impact seems to be in engineering curriculums in the areas of computer science, computer engineering, and electrical engineering. These academic disciplines have been the genesis of basic concepts in digital hardware and software systems and presently prove to be the focus point in the university community for introducing new trends such as those exemplified by microprocessors. Because of the general shift to digital processing techniques and the particular interest in using microprocessors it may seem logical that a fundamental alteration in an academic curriculum may occur [6]. The exact form this will take is still an open question. However, it is the contention in this article that since the microprocessor and microcomputer are but a subset of the general and familiar class of Digital Systems that is is not necessary to teach formal lecture courses on microprocessors. All of the fundamental material needed to understand the conceptual behavior of these devices and systems should already exist in a modern Computer Science academic program. This would include switching theory, programming methods and languages, system architecture, and other such topics. What is needed is a well developed laboratory program which will support and complement the formal lecture courses so that the conceptual information will have a meaningful practical base. In addition, such a laboratory will give the student an opportunity to explore the peculiarities of various microprocessor devices, acquire hands on experience with different microcomputers, and in general accrue the benefits normally provided by a solid laboratory experience [7,8]. The Microprocessor Laboratory described here is intended, in addition to promoting graduate research as mentioned earlier, to initially support two formal laboratory courses and be an adjunct to a lecture course in digital systems design in the University of Connecticut Computer Science program, and support a laboratory course in digital control systems and be an adjunct to existing laboratory courses in the Electrical Engineering program. It is hoped that as the situation develops this facility will also support other undergraduate and graduate activities.

Practical Work Experience in a University

One fundamental influence of microprocessors on academic activities is the possibility of exposing the student to a "real world" experience which will truly prepare him for the type of work he will encounter and be required to perform in industry. Certainly there seems to be a continuing complaint that the university community is not responsive to the needs of industry in preparing their graduates to take their place in society as productive engineers or computer scientists. This is a complex question with a balance needed between teaching fundamental, general concepts which will serve the student throughout his career and providing the student with specific skills which meet the current and possibly transitory needs of industry. It would seem that a productive answer to this conflict is to build a strong laboratory program which serves to educate the student in specific skills by orienting the experimental work towards current industry activities while still stressing the fundamental concepts in the lecture courses. With the increasing emphasis on digital technology, digital systems, and digital information processing methodologies, the microprocessor seems to be a tailor-made avenue for creating a strong laboratory program which will bridge the abstract needs of a strong academic curriculum and the current technical emphasis in industry. As has been discussed by others [9], this has not been feasible due to economic considerations until quite recently. But, as industry can now consider introducing digital techniques into previously inaccessible areas due to the low cost and small size of microcomputers, it is now possible (even with present budgets) to permit the student to have access to real time, hands on powerful computers in their experimental work. Also, the general nature of the microprocessor should permit rapid restructuring of a laboratory program to reflect current industrial activities.

Microprocessor Laboratory Organization

One paramount disadvantage of microprocessors and their related devices is that a significantly higher level of support is required from a laboratory facility to permit an arbitrary user to utilize them conveniently [10]. Unlike experiments dealing with SSI and MSI logic devices [7] where simple breadboards and a stock of integrated circuits will suffice, the microprocessor introduces a high level of complexity into the situation. Complex breadboards are required for we are now dealing with system integration considerations. Special networks such as ROM programmers are necessary to consider special purpose applications and to bypass software loading problems. A full range of software support is necessary to permit software design to proceed in a reasonable manner. A variety of peripherals such as mass storage systems, computer terminals, and other such devices are needed for stand alone, real time operations or experiments. Coupled with this is the fact, as discussed above, that in a university situation we are dealing with not only a multi-use, but a high-use environment. Assuming no duplication of facilities (which is a reasonable assumption with today's economic condition) the laboratory discussed here may in some typical week support eight sections (12 students per section) of formal undergraduate laboratories, be a medium for carrying out senior independent study projects (10 per semester), and be an experimental facility for graduate research (5 to 10 per year). So not only must it switch function from day to day (a microprogramming operation one day and a digital control lab the next, as an example) but in excess of a hundred people may be using the laboratory in any given week.

A solution to meet these requirements is the laboratory described here. It is organized into four levels of operation which functionally consist of: (1) A PDP 11/20 microcomputer system to permit software design, development and experimentation to occur in a comfortable environment; (2) a PDP 11/10 minicomputer with a data link to the PDP 11/20 system to permit hardware design, development, and experimentation to take place in a comfortable environment; (3) A microcomputer system integration mode whereby the user builds a microcomputer by interconnecting and using standardized, functional, digital system modules; (4) A breadboarding activity whereby the user may design and build a prototype of a microcomputer or digital system module. The system integration mode will permit the laboratory user to bypass device characteristic problems and concentrate on system level concepts while the breadboarding level will permit a given individual to explore device level characteristics in a given design problem.

Dual PDP 11 Minicomputer Support System

By "comfortable environment" in the minicomputer activity it is meant that a means for file organization, rapid program development, operating system support,

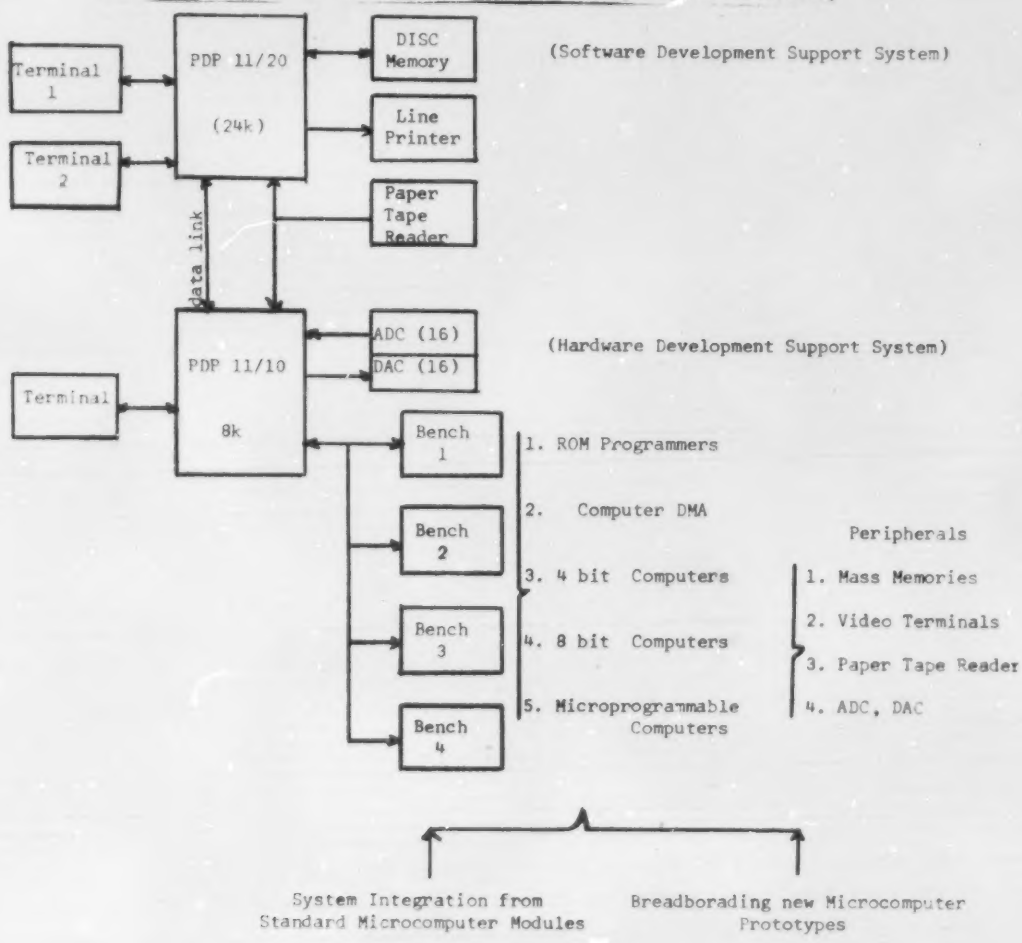
easy hardware interfacing, ROM programming, time shared systems for multiusers, and a number of the other necessary support features in a high-use facility will be provided by the dual PDP 11 minicomputer system. A block diagram of this minicomputer support system is shown in Fig. 1. The PDP 11/20 segment is the larger element in the dual configuration as it contains the features necessary to a high level of software support. It contains features for user program and data storage, generation of hard copy output, an operating system to aid in program development, cross assemblers for the microcomputer systems, high level language support for the microcomputers, and other such options. The PDP 11/10 has a data link to the PDP 11/20 so it may access peripherals in that system and to have access to a high speed medium to transfer object program modules from the support software in the PDP 11/20 to the PDP 11/10. As shown in Figure 1, the PDP 11/10 is directly interfaced to work benches in the laboratory via a direct memory access feature in the microcomputer and also act as a high level test and debugging aid in microcomputer system development. The PDP 11/10 also controls ROM programmers for firmware generation, provides analog to digital and digital to analog conversion features, and may serve as a powerful control node in multiprocessing. The two minicomputers are located in adjacent but physically separate rooms. This permits various combinations of software development and hardware experimentation to take place simultaneously thus satisfying some of the requirements of the multi-use, high-use environment.

Microcomputer Development and Experimentation

As illustrated by Figure 1, the UCONN Microprocessor Lab permits a variety of operational configurations for either developing or experimenting with microcomputer systems. Some of these various modes of laboratory operation may be summarized as:

1. Transfer object program modules from PDP 11/10 to a microcomputer memory.
   a. Read/Write semiconductor memory via a microcomputer DMA channel
   b. Erasable programable read only memory
   c. Programable read only memory
2. Interaction between the PDP 11/10 and a microcomputer
   a. PDP 11 testing a microcomputer
   b. PDP 11 and a microcomputer interacting in carrying out a task
3. Interaction between the PDP 11/10 and up to four microcomputers
   a. Permits immediate experimentation in networking, parallel processing, distributed processing, process control, and other multicomputer configurations

Fig. 1 A Block Diagram of the Dual PDP 11 Minicomputer System Which Supports the Microprocessor Level Activity in the UCONN Microprocessor Laboratory



4. Stand alone microcomputer

   a. Object programs loaded via DMA, ROM firmware, or paper tape reader

   b. System support performed by standard microcomputer peripherals such as a mass memory system, a low cost video terminal, and analog to digital and digital to analog systems

5. Microcomputer arrays

   a. Object programs loaded via DMA, ROM firmware, or paper tape reader

   b. Microcomputer support by standard microcomputer peripherals

   c. Microcomputer interconnection via standard I/O bus structure

6. Microcomputer Hardware System Design and Development

   a. Integration of high level off-the-shelf functional modules

   b. Breadboarding new prototypes by starting at the device level

7. Microcomputer Software System Design and Development

   a. Use the PDP 11/20 system for source program design

   b. Use the PDP 11/10 system and some choice of modes one through six for object program testing

8. Investigation of Microprogramming Systems

   a. Use the PDP 11/20 for high level language microprogramming

   b. Use microprogramable microprocessors for test and development

9. Use of Read-Only Memories in combinational and Sequential Logic Design

   a. Use the PDP 11/10 to program ROMs to be used elsewhere as functional units in logic networks

10. Off-the-Shelf Firmware Development

   a. Use the entire laboratory as a vehicle to design and test off-the-shelf firmware. This may be microcode or macrocode which a given user may draw from a parts cabinet as per his individual needs. Examples of this application might be emulation, process control operations and other such applications

11. Undergraduate Digital Systems Laboratory: formal experiments

   a. Computer Interfacing Experiments
      1. Microcomputer to PDP 11/10
      2. Microcomputer to Microcomputer
      3. A/D and D/A concepts

   b. Hands on experience with Microcomputers
      1. Stand alone systems as per mode four

   c. Hands on experience with microprocessor devices

   d. Hands on experience with microcomputer peripherals

   e. Hands on experience with semiconductor memories

12. Undergraduate Digital Control System Laboratory: formal experiments

   a. Various combinations of the laboratory shown in Figure 1 may be used to structure control systems experiments by adding appropriate physical systems to the laboratory along with appropriate transducers to interface to the non-digital world

13. Senior Independent Projects

   a. Microprocessors seem to offer a unique opportunity to universities to permit students to experiment with non-trivial problems in order to achieve a practical experience in an academic situation which will prepare them for industry

Other possible uses of this laboratory facility may of course be proposed but it is hoped that the above list indicates the flexibility of such an approach and illustrates how it answers the needs of a multi-use lab in a university. A key element in this entire approach is the existence of a multitude of flexible microprocessor based systems which permit an individual to consider the implementation considerations as well as the abstract concepts associated with a given digital system.

Prototyping of Microprocessor Functional Modules

The second foundation stone of the laboratory (if the dual PDP 11 support system is the first) is the existence of a variety of microcomputers and standardized, functional microprocessor oriented modules from which to construct different microcomputer systems or networks of microcomputers. Although a number of commercial microcomputer kits are available at present (Spring 1976) which include both the electronics and a printed circuit board, this was not the case when this lab was first started in the Fall of 1974. This necessitated an in-house design and development activity to develop the requisite systems. Three basic decisions occurred at that point in time. These were: (1) to build a capability to experiment with a variety of microcomputers; (2) to define a set of functional, microprocessor oriented, digital systems modules which were directly interchangable between different microprocessors; (3) to derive an inexpensive, reliable, and flexible method for prototyping the functional modules to be used in the laboratory.

One typical limitation in a university is a lack of personnel and facilities for building new electronic systems. To do it correctly would ideally require a team consisting of the system designer, packaging engineer, draftsman, printed circuit and assembly, and a skilled technician to construct and test the completed system. In an academic environment, a typical design team might consist of a single student responsible for carrying out all phases of a given project. Also the budget would normally be stringently limited. To provide a possibly inexperienced student with a situation which would help him physically assemble a low cost solution to a design problem, a set of versatile breadboarding elements were selected and a formal methodology was derived which when followed leads to a physically reliable prototype.

The various systems of interest were analyzed and it was decided that seventy two edge connections would suffice for signal and power connections for the physical board on which the electronics would be mounted. Because of the variance in package size (14 to 40 pin-dual in line) in modern microprocessor oriented devices,

no commercial breadboard seemed appropriate. Therefore, it was decided to choose a set of breadboarding elements which could be optimally configured to the function. A set of four elements were chosen from which all of the microprocessor functional modules have been prototyped. These are:

1. A 9" x 4.5" Vector perforated board with holes on .100" centers and 72 printed circuit edge connections
2. A stock of sixteen to forty pin dual in line, wire wrap sockets
3. A two layer, 9" Rogers Mini-Bus, with solder connections, for power distribution and noise suppression
4. Epoxy glue to hold the sockets to the board

A picture of these elements is shown in Figure 2. The student may then configure the board to the function with the total cost of the packaging hardware being under twenty five dollars. He may fit the different size sockets on the board in an appropriate fashion and be exposed to some experience about package design in the process. A key element in this approach is the Rogers Mini-Bus [11]. It has three outstanding features which insure the success of this approach. This device: (1) Provides a third dimension for distributing power and grounds on the Vector board; (2) Suppresses switching noise on the power lines and permits proper termination points for decoupling capacitors if needed; (3) Imposes a physical organization on the board layout which promotes a minimal noise system, leads to a direct mapping to an eventual printed circuit form, and guides the student into an acceptable board layout. The first two features are technical considerations whereas the third is critical in helping the untrained student to avoid irrevocable layout errors.

The present experience in this laboratory has proven this approach to be quite successful. The students are able to translate their conceptual designs into a working piece of hardware within a comfortable time frame. A wide variety of operational microprocessor functional modules have now been generated by this approach.

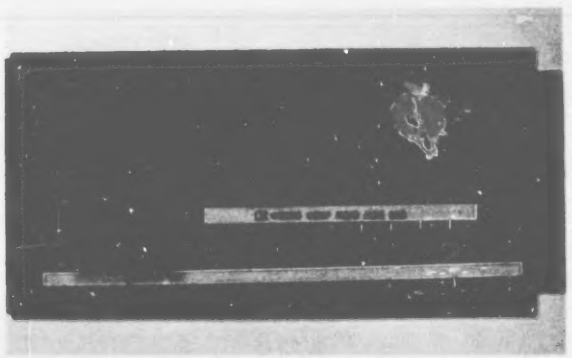


Figure 2 Breadboarding devices from which to construct microcomputer modules

Summary of Microprocessor Functional Modules

It seems apparent that no single microprocessor will be the answer to the broad spectrum of application areas presently being envisioned for microprocessors. In addition the microcomputer system configuration will vary according to the requirements of a given problem. Therefore, it is felt that having the capability to draw upon different microprocessors will promote the needs of the multi-use laboratory. In addition, different memory, peripheral, control, and power configurations will be required as problems vary. The functions normally encountered in typical digital systems were tabulated and an attempt has been made to design and develop a set of off-the-shelf functional modules which could be used in any microcomputer system in this lab when required. A listing of the present functional modules is:

1. 4 bit Microprocessor card (4040)
2. 8 bit Microprocessor card (8080A)
3. 2k x 8 Read/Write Memory card (2101)
4. 2k x 8 Read/Write Memory card (2102)
5. pROM Memory card 2k x 8
6. Erasable pROM Memory card 2k x 8 (8704)
7. Erasable pROM Memory card 2k x 8 (1702)
8. 8 bit system control card (8080A)
9. Analog to digital convertor card
10. Digital to analog convertor card
11. 53 function electronic keyboard
12. Low cost video terminal
13. Standard UART based computer terminal card
14. Low cost peripheral mass memory
15. High speed paper tape reader interface card
16. 16 bit microprogramable microprocessor card (3000)
17. 512 x 40 microprograming read/write memory card
18. 512 x 40 microprograming ROM card
19. PDP 11 interface card
20. Standard chassis

Figure 3 and Figure 4 present pictures of functional modules two and three. These pictures will also illustrate how the packaging elements shown in Figure 2 are assembled to realize a working hardware prototype. All of the memory cards have an internal logic network which permits them to be assigned to any address space from zero to 64k. In addition, all cards have a standard input and output function which (1) defines the logical behavior of each signal on each input or output connection of each card to be identical and (2) defines the circuit characteristic of each connection on each card to be a standard current load of a TTL lowpower Schottky gate for each input and a logically controlled tri-state gate for each output. This permits logical control of the output of each card and permits any given system to consist of up to sixty four cards with all of the fan out considerations being incorporated into the card input/output design itself.

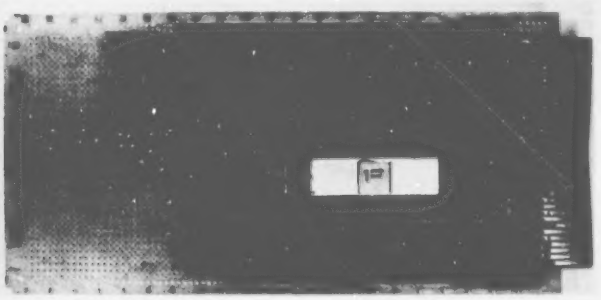


Figure 3 A picture of the prototype for the 8 bit microprocessor card

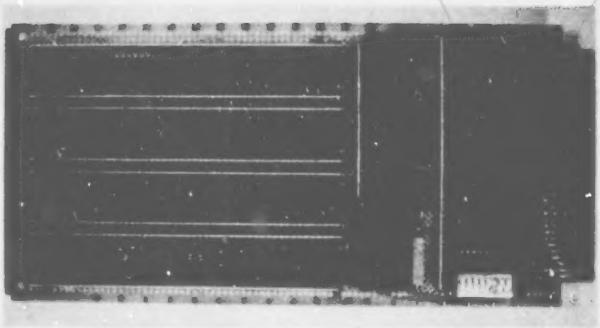


Figure 4 A picture of the prototype for the 2k x 8 Read/Write memory card

Standard Chassis

In addition to the input/output circuit characteristics of a functional module, a standard data, control, and power bus configuration is implemented in the form of a standard computer chassis with a preassembled backplane. Each chassis is capable of holding ten cards with any card being placed in any arbitrary slot. Careful attention was made to formatting the power, data, and control lines to permit this feature. Single direction data lines were used to simplify data bus circuitry and enough flexibility exists in the control bus to realize different microcomputer systems without altering the backplane wiring. A number of these standard chassis modules were constructed and support an impressive variety of projects. This has the added benefit of avoiding a possible time consuming stage in system development and imposes compatibility between systems developed by different students. A picture of the card chassis is shown in Figure 5. The backplane wiring consists of a device called the Rogers Bus strip. This is used to minimize cost and construction time and promote physical reliability.

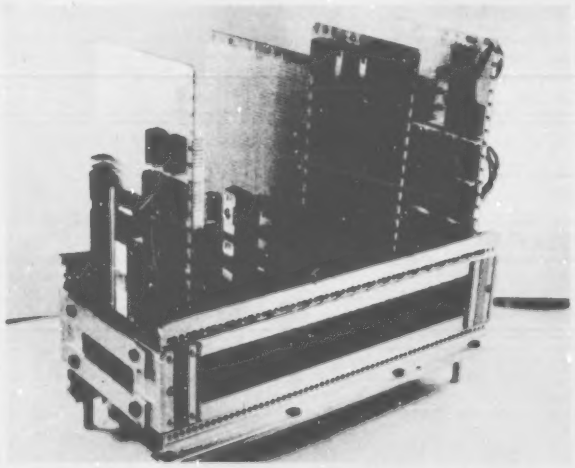


Figure 5 Standard microcomputer chassis with prewired backplane. This example contains a processor card, ROM card, read/write memory card, and teletype card.

Conclusions

It is still too early to tell if the facility discussed here is the entire answer to the requirements of a high use, multi-use microprocessor laboratory in a university environment. However, the experience and results to data have been encouraging in that the design and development of a set of complex microprocessor based systems modules has been successfully carried out using the lab organization described here. This has provided us with a means for moving on to the next stage whereby the range of activities described earlier may now be investigated.

A number of definite results have been achieved. It does seem feasible to construct a laboratory of this type in a reasonable time frame under reasonable budgetary considerations. It does seem possible to expose undergraduate and graduate students to significant design experiences during their academic career. The training contained in a modern Computer Science curriculum does seem to provide a solid conceptual base which permits the student to comfortably design and work with microcomputer systems. The microprocessor, and related devices, does seem to provide a mechanism to study an impressive variety of problem topics. We've looked at real time speech synthesis, data acquisition systems, process control, microprogrammed systems, and other such diverse topics. The facility does seem to be multi-use with considerations now underway for incorporating people from Chemical Engineering and Mechanical Engineering into its operation. It is possible to construct physically reliable, low cost prototypes of design solutions in an academic program. These and other results indicate that the current excitement about microprocessors is well founded.

The next phase of this project will explore whether it is possible to permit the untrained computer user to define and easily use microcomputer systems. This will now be possible with the present setup. To date a strong emphasis has been placed on building a set of hardware tools and the dual minicomputer support system. Now the system integration problems can be measured and the software tools needed will be studied. We can then attempt to permit an individual from outside the Computer Science field to use the microprocessor lab and determine what tools and training will be necessary for him.

References

1. R. L. Barron, "Learning Networks Improve Computer Aided Prediction and Control", Computer Design, Vol. 14, No. 1, Aug. 1975.

2. M. E. Conway, "A Multiprocessor System Design", Proceedings of 1963 AFIPS Fall Joint Computer Conference, pp. 139-146.

3. D. Graupe, W. K. Cline, "Functional Separation of EMG Signals via ARMA Identification Methods for Prothesis Control Purposes", IEEE Transactions on Biomedical Engineering, Vol. BME-22, No. 3, May 1975, pp. 225-229.

4. A. R. Ward, "LSI Microprocessors and Microcomputers: A Bibliography", Computer Vol. 8, No. 7, July 1974.

5. A. R. Ward, "LSI Microprocessors and Microcomputers: A Bibliography continued", Computer, Vol. 9, No. 1, Jan. 1975.

6. T. L. Booth, B. J. Carey, "Impact of Micro/Minicomputers on Electrical Engineering Curriculums",

Proceedings of COMPCON, Fall 74, IEEE Cat. No. 74CH0869-8C.

7. T. L. Booth, et. al, "Digital Systems Laboratory Courses and Laboratory Developments", A Report of the COSINE Committee of the Commission on Education of the NSF, March 1971.

8. J. Grason, D. Siewiorek, "Teaching with a Hierarchically Structured Digital Systems Laboratory", Computer, Vol. 8, No. 12, Dec. 1975.

9. H. S. Stone, "Parallel Computers", Chapter 8 in Introduction to Computer Architecture, edited by H. S. Stone, Science Research Associates Inc., 1975.

10. B. J. Carey, "Standard Microcomputer Modules", First Annual IEEE Workshop on Applications of Minis and Micros, Nov. 1975.

11. H. Grossman, B. Carey, R. Jodoin, "Mini-Bus, The Printed Circuit Board Bus Bar", Application Note, No. 1984, Rogers Corp., Rogers, Ct., 1976.

A MICROPROCESSOR LABORATORY COURSE

John F. Wakerly
Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California

1. INTRODUCTION

The microprocessor laboratory course in the Electrical Engineering Department at Stanford University is a 3-unit, one quarter (10 weeks) course with the following prerequisites from the existing undergraduate curriculum:

1) Introduction to Computer Organization, Machine and Assembly Languages. Programming in a higher-level language is assumed as a prerequisite. In this course the student learns basic computer organization, assembly language for a particular minicomputer, basic I/O programming, and simple data structures. With this background, the student can start writing microcomputer programs in the first week of the lab course.

2) Digital Computer Organization. This course covers basic switching theory and logic design, logic families, computer arithmetic, processor organization, memory, input/output, and microprogramming. With this background, the student can easily grasp typical microcomputer system organization and he can easily design support circuits for a microcomputer system.

3) Digital Logic Laboratory. In this course the student gains hands-on experience with TTL SSI and MSI integrated circuits. This course is necessary since each student in the microcomputer lab is given a set of microprocessor and support chips and required to build his own system. We don't want to spend time teaching the students things like which end of an IC is up.

With the prerequisites shown above, each student can build and program his own microprocessor system during the first four or five weeks of the course. The remainder of the quarter is used for a final project.

The course includes both lectures and lab assignments. These are described, along with the lab equipment, in the following sections. The course serves 25 to 30 students per quarter and is offered two or three quarters per year.

2. LECTURES

Two 50-minute lectures per week are given during the first four or five weeks of the course. The lectures supplement and reinforce the intensive reading assignments of the first few weeks of the course. During this time the student learns the specifics of a particular microprocessor (the 8080) and its development system. Approximately one lecture is devoted to each of the following topics in the order given:

Overview of 8080 architecture, especially from the programmer's point of view
8080 assembler and instruction set
Microcomputer development system
8080 logical and electrical specifications
8080 support chips and memories
Design of a small 8080 system
Interrupts and DMA in the 8080
PIAs and UARTs

Reading assignments are taken from Intel, 8080 System User's Manual; Intel, 8080 Assembly Language Reference Manual; and Intel, MDS-800 Microcomputer Development System Operator's Manual.

During the second half of the quarter, lectures are reduced to once per week. Each lecture surveys the characteristics of a different microprocessor. The lectures are important as a continuing point of contact with the lab students for problems, announcements, and so on, as well as for the lecture material itself.

3. LABORATORY ASSIGNMENTS

There are four laboratory assignments given during the first five weeks of the course. Each student works by himself, and all students do similar projects. The assignments are described below. Although they are described in terms of the Intel 8080, they can easily be adapted for any microprocessor. Lab assignments and some supplementary material are taken from a lab manual, John F. Wakerly, Logic Design Projects Using Standard Integrated Circuits, John Wiley, 1976.

Lab #1 - Development System Familiarization

This is a simple exercise to quickly introduce the students to the operation of the Intel Intellec® MDS-800 Microcomputer Development System that supports most of the programming in the lab. The students are given a hand-out that takes them step-by-step through the entry, assembly, loading, modification, and execution of a simple program. The assignment must be done by the end of the first week of class; it takes about an hour of lab time.

Lab #2 - Simple 8080 Program

The students are asked to write a program for a small 8080 system. The target system has the features shown in Figure 1. The reset pushbutton initiates an 8080 reset; the interrupt pushbutton initiates an 8080 interrupt. The system has 256 bytes of PROM but no RAM at all. There is a single latched output port that drives eight lamps.

Each student must write a short assembly language program that does something interesting with the pushbuttons and lamps. The student can develop his own program idea or use one of our suggestions such as an event counter, reaction timer, or lamp ping-pong game. In any case, the program must use no RAM and must fit in 256 bytes of PROM.

Each student must punch his program on cards, assemble it on the microcomputer development system, and execute it on the microcomputer development system using our general-purpose I/O box for the lamps and switches. The project is due after the second week of class. Students must turn in a commented listing of their program, operating instructions, and a paper tape to be tested by the teaching assistant. Last quarter a typical student spent 2 hours at home and 2 hours in the lab on this assignment. A few students spent half this time and a couple spent two or three times this amount.

Lab #3 - 8080 Timing Waveforms

Each student is given a breadboard and a set of parts and asked to build the circuit shown in Figure 2. He must predict, observe, and sketch the waveforms on Ø1, Ø2, SYNC, $\overline{WR}$, and DBIN for one instruction cycle (the system is executing the same instruction, RST 7, continuously). He must also explain the behavior of the address and data lines. Finally he is asked to connect the WAIT output to the RDYIN input and explain the effect on the circuit's operation (this inserts one wait state for each memory reference).

The students must turn in a logic diagram of their circuits, timing diagrams, and answers to questions. Last quarter the time spent on this project varied from a minimum of two hours at home and one in the lab to five hours at home and five in the lab. The average was six hours total, with the distribution between at home and in the lab varying with each student's work habits. The project is due after the third week of the class.

Lab #4 - Small 8080 System

Each student is given parts to design and build a small 8080 system that will run the program he wrote in Lab #2. A typical solution takes only 6 chips -- an 8080, 8224 clock generator, 8228 system controller, 1702 PROM, 8212 output latch, and 1/2 - 74LS74 interrupt pushbutton edge detector. Eight LEDs and a few discrete resistors and diodes are also required. Negative voltages (-5 and -9) are derived from -12 with zener diodes and resistors. The students are also required to design and implement a single-step circuit for the 8080. A clever student can do this using only the remaining 1/2 - 74LS74.

Each student must turn in an industrial-standard complete schematic and theory of operation for his system, as specified in the lab manual. He must also give the teaching assistant a demonstration of his working system.

The time for this project also varied widely, with 3 hours total claimed by the fastest student and 13 by the slowest! The average was about 9 hours, again with the distribution between the lab and home depending on the student. The project is due near the end of the fifth week of the course.

After the first four labs, five weeks remain for the students to do final projects. Most students work individually. The project may be a single five-week project or a combination of two shorter projects. Project ideas can be taken from the lab manual or the student can come up with his own idea. In either case, the student must turn in a proposal detailing the project idea, complexity, and parts and equipment required. The instructor's feasibility evaluation at this point is very important in terms of both parts and equipment availability and time required. A word of warning -- do not accept project proposals calling for parts not already in stock, and do not accept over-ambitious proposals from enthusiastic students unless they show that they can find a reasonable subsystem to complete and demonstrate if they run out of time.

The student must turn in commented program listings, theory of operation, block diagrams, and industrial-standard schematics for their projects. All documentation plus a project demonstration are due at the end of the last week of class, before finals. This is as much for the student's good as the instructor's. No "incompletes" are given. If a student cannot finish in time he may be allowed to finish a subset of his proposed project and take a reduction in units for the course.

4. DEVELOPMENT SYSTEM

The microprocessor lab is centered around an Intel Intellec® MDS-800 microcomputer development system. We have the following configuration pictured in Figure 3:

CPU with 48 Kbytes RAM
6 Kbytes PROM
CRT terminal
Teletype
Floppy disk with Intel ISIS operating system
Card reader
Papertape reader/punch
120 char/sec printer
PROM programmer
General-purpose I/O box
8080 in-circuit emulator

This configuration is just about as complete as anyone could want, and all of the equipment is used in a typical day of operation. Actually, we have two identical systems, except that the printer and papertape punch are shared by means of a switch.

Most programs are written in assembly language, punched on cards, and assembled and printed under the control of the floppy disk operating system. We have a truly resident, stand-alone system; a cross-assembler and a PL/M compiler are available on the campus facility's IBM 370, but they are not used for most student programs. Paper tape is not needed except as a backup medium and for object tapes produced by the IBM 370. The development of hardware and software for downloading programs from the IBM 370 via a telephone link is near completion.

Programs can be tested by running them on the development system. A general-purpose I/O box (Figure 4) provides visual display of four 8-bit output ports, and switches for one 8-bit input port that can be accessed by the user program. The signals of the four output ports and four input ports are buffered by optical isolators and made available at 25-pin connectors on the I/O box. This allows a student to design a system that uses the CPU and memory of the development system but communicates via the 25-pin connectors to special-purpose I/O circuitry on his breadboard.

Programs for stand-alone systems built on the students' breadboards can be "burned" into PROMs using the PROM programmer. The PROMs can be erased with an ultraviolet lamp.

The 8080 in-circuit emulator has a cable with a 40-pin plug that is plugged into a student's system in place of his 8080. This allows the user's system to be single-stepped and observed at the development system console, and it allows memory and peripherals of the development system to be selectively made available to the user's system. Since we are wary of students making any direct electrical connections to the development system, we only allow use of the in-circuit emulator by advanced students in advanced projects.

The 6 Kbytes of PROM in the system are available for permanent system programs. Currently less than 1 Kbyte is used, for the card reader driver and down-loading software.

5. HINTS FOR DEVELOPMENT SYSTEMS

Although our large configuration is very nice, not all of it is necessary for a decent lab. If assembly can be done on a larger system, then a CPU with 16 Kbytes of RAM, a CRT terminal, a papertape reader/punch, and a PROM programmer will suffice. Some kind of general-purpose input switches and output lamps are also strongly recommended for aiding program testing.

If assembly is to be done on the development system, then a fast printer will be needed if the lab is to serve more than a couple users. A card reader is also strongly recommended, since the use of the terminal for text-editing takes a very large amount of system time.

A floppy disk operating system will double system throughput in most cases by eliminating the need to load system programs from paper tape. It will obviously aid greatly in the development of larger programs.

A luxury that may be well worth the price is to have two identical systems. In addition to providing double the system time, having two systems greatly simplifies maintenance and reduces the panic when failures occur. However, the word identical should be stressed. There are actually more reliability problems if a lot of interfaces and devices must be moved from the primary system to a smaller back-up when the primary fails.

The policy of what hours the development systems (and the rest of the lab) are available is up to the school. At Stanford we make the lab available to lab students weekdays from 8:00 - 5:00. Trusted students working on special projects have keys to use the lab at other times if they wish. With 30 students taking the lab course and over a dozen more on special projects, the available time on the two microcomputer development systems has been more than adequate; sign-ups for time have been unnecessary.

6. BREADBOARDS

The breadboard kits used in the microprocessor lab were designed and built at the Digital Systems Laboratory for a parts cost of about $200 each. Each kit consists of an attache case and three modules, as illustrated in Figures 5 and 6. The modules are mounted on two aluminum rails that distribute +5 volts and ground throughout the system. A small area of the suitcase is available for storage of parts and small tools. The breadboard unit has no power supply; power must be supplied from an external source as indicated in the module descriptions below.

I/O Connector Module

A. Binding posts

Two binding posts on the module are intended for supplying +5 volts and ground to the system. The RED binding post is connected to the +5 rail by a printed circuit trace under the board, and the BLACK binding post is connected to the ground rail.

Three additional binding posts with no internal connection to the breadboard are provided. These binding posts are intended for supplying +12 volts (YELLOW), -12 volts (WHITE), and ground (BLACK) to the breadboard. When they are used for this purpose, the user must connect the circuits that require +12 volts and -12 volts directly to the yellow and white binding posts, and the black binding post must be connected to ground (the other black binding post).

B. Pushbuttons

Two debounced pushbuttons P1 and P2 are provided. Four eyelets provide TTL signals that indicate the state of the pushbuttons. P1 H is HIGH when P1 is depressed, LOW otherwise; P1 L is the complement of P1 H. The signals for P2 are similar. P1 H and P2 are indicated by red dots next to their eyelets; P1 L and P2 L are indicated by black. A 7404 is used as the debouncer.

C. Socket strip

IC's and wires may be inserted into the eyelets of the socket strip. The socket strip has 128 rows of five electrically common terminals (holes). Thus, when a 14-pin or 16-pin IC is inserted across the center of the strip, each IC pin will be in common with four holes, facilitating point-to-point wiring. Wider IC's (such as 24-pin and 40-pin packages) will leave fewer holes available for wiring.

The two outermost columns of holes are connected to +5 volts and ground, as indicated in Figure 5. The two inner columns are not connected and are not used. Throughout the breadboard, +5 volt columns of holes are marked with RED dots, ground columns with BLACK.

D. 25-pin connectors

Two 25-pin connectors J1 and J2 are provided for interfacing with external devices. The pins of J1 and J2 are internally connected to the rows of socket strip. Thus the user may connect wires to the appropriate rows of the socket strip to establish a connection with an I/O device. These

connectors can be used to connect to CRT terminals or to the development system I/O box.

Socket Strip Module

A. IC strips

There are five strips for plugging in IC's. Each strip has 128 rows of five electrically common holes. These holes are not connected internally to anything else.

B. Power and ground strips

Six narrow strips are used to distribute +5 volts and ground. Each strip has two columns of holes, one for +5 volts (RED) and one for ground (BLACK), as shown in Figure 5. The power is obtained from the aluminum rails.

72-Pin Connector Module

A. Binding posts

There are three binding posts similar to the three floating binding posts of the I/O connector module. They are not connected internally to anything else in the breadboard. The posts can be used to bring other voltages onto the breadboard or for any other purpose.

B. Socket strip

This strip is identical to the socket strip on the I/O connector module.

C. 72-pin connectors

J3 and J4 are standard connectors that accept Digital Equipment Corporation double-height modules or Unibus cables. Like-numbered pins of the two connectors are internally connected together and to the rows of the socket strip. A lamp and switch module (illustrated in Figure 7) can be plugged into one of these connectors.

7. HINTS FOR BREADBOARDS

For the school planning to buy breadboards or to design and build their own, there are a number of factors and features to consider.

Number of Breadboards

It is most desirable to have a separate breadboard for each student. In this way, projects need not be limited to those that can be built and disassembled in a single lab session. However, if the number of breadboards is limited, it is still possible to design meaningful projects for 4-hour lab sessions.

Power Supplies

If just a few breadboards are needed, it is just as well to include a power supply with each. However, an OEM power supply that delivers 3A at +5 volts and .5A at ±12 volts costs about $80. If individual breadboards are being provided for each student, it is better to build the breadboards without supplies and provide supplies in the lab that can be shared like test equipment and other lab facilities.

Portability

Regardless of whether or not each student has his own breadboard, almost all project debugging will have to be done in the lab where the test equipment, PROM programmers, and other facilities are located. However, if each student is assigned his own breadboard, portability allows wiring to be done at home in the evenings instead of during limited daytime lab hours, a big advantage for most students. There is no security problem since each student is responsible for a particular breadboard whose number is recorded.

Pre-wired Components

Some commercially available microprocessor breadboards have a microprocessor, support chips, and memory pre-wired on a PC board. This reduces wiring complexity and can be a big advantage if the students have limited access time to the breadboard. On the other hand, it reduces flexibility tremendously. A small system takes only an hour to wire up from scratch and breadboards that allow this have the advantage that they can be used with any present or future microprocessor.

Socket Strips

Breadboard socket strips such as the ones shown in Figure 6 have become quite standard and are available from a number of manufacturers. A minimum of four strips should be provided on microprocessor breadboards. Strips that distribute power and ground should be color coded to avoid costly wiring mistakes.

Connectors

Connectors are a very important part of microcomputer breadboards. As a minimum, there should be two standard 25-pin connectors for communicating with terminals or the I/O ports of the development system. Additional standard 44-pin or 72-pin connectors allow larger cables or PC boards to be plugged in. The PC boards can contain lamps and switches, memories, or other subsystems.

Lamps and Switches

Lamps and switches are a great convenience in any breadboard. Eight switches, a dozen lamps, and four debounced pushbuttons are sufficient for most applications. Not all projects require lamps and switches, but two pushbuttons and a toggle switch (for reset, step, and run/step) are a good minimum. Additional switches and lamps can be provided by outboard modules (Figure 7) that are shared just like power supplies.

IC Protection

ICs that cost over $5 should be mounted in sockets to protect their pins if they are to be used with breadboards. We have found that Augat solder-tail sockets are the best but more expensive choice for this application. All of the ICs visible in Figure 8 are mounted in such sockets.

IC Labeling

Labeling the pins of LSI ICs as shown in Figure 8 is very effective at both speeding up wiring and reducing errors. A master label can be typed and then reduced with a Xerox machine or other means; labels are pasted to the backs of ICs with rubber cement. Labels for SSI and MSI ICs are sold by E & L Instruments.

Wire

Multiple colors of #22 solid hookup wire should be provided for use with the breadboards. Students should be encouraged to wire neatly and use a color code, as in Figure 8.

8. TEST EQUIPMENT

A number of test instruments are needed for observing and debugging the performance of microprocessor circuits. Oscilloscopes are the most important instruments in a digital lab. A 50 MHz dual-trace scope such as the Tektronix 455 is fine for this application; two are used in Stanford's lab.

Logic analyzers are excellent test instruments for debugging large synchronous sequential circuits and microprocessor systems. A basic logic analyzer has a clock input and 8 to 16 data bit inputs. At each clock pulse, the data inputs are strobed and stored in a digital memory. The stored input words are then displayed on an oscilloscope screen, either as binary digits or in a timing-diagram format. From 16 to 256 or more words of memory are provided, so that a long record of data input words can be observed. Triggering circuits are usually provided so that storage and display of data is not started until a selected input word is observed. The triggering point can be moved in time so that a number of words both before and after the triggering word can be displayed. Some logic analyzers also have the capability of triggering on asynchronous input pulses, independent of the clock input. This feature is especially useful for detecting and diagnosing system "glitches." Stanford's lab has one of the first commercially available logic analyzers, an HP 1601L; newer models are nicer.

Hand-held logic probes and pulsers are very useful debugging devices. A typical logic probe can be touched to a signal line to indicate whether it is at logic 0, logic 1, or an intermediate or floating level. Some probes have built-in latches to catch pulses too short to give a normal visual indication. A logic pulser emits a short, high-energy pulse that overrides any level being maintained by standard TTL outputs. This is primarily useful for debugging printed circuit modules, since it allows test signals to be injected without breaking existing connections. For breadboard debugging, a pulser is less important since connections can be broken and test signals injected by hand fairly easily. We use HP logic probes and pulsers in the microprocessor lab at Stanford.

Digital multimeters are found in most labs, but they are not essential to a microprocessor lab. Voltage measurements can be made with an oscilloscope, and resistance and current measurements are not often needed.

9. CONCLUSION

The microprocessor lab course at Stanford has been very successful, from both the viewpoints of the faculty involved in the course and the reactions of the students who have taken the course. The approach of designing a lab for only students with strong prerequisites in both logic design and programming has been the appropriate one, at least for the student population at Stanford.

Although the capital equipment outlay for Stanford's lab was substantial, a decent development system and peripherals can be purchased for $10K to $15K, and hardware breadboards can be built for a few hundred dollars each. With the price of LSI ICs continuously decreasing, the cost of the ICs for a microprocessor lab is no more than for an SSI and MSI digital logic lab.

We gratefully acknowledge a number of sources of support for the equipment for our lab: the General Electric Foundation, the National Science Foundation, Intel Corporation, and Motorola. Professor Edward J. McCluskey deserves credit for pushing on both the department and the author for development of the lab, and Dr. Victor H. Grinich aided greatly in the lab's development.

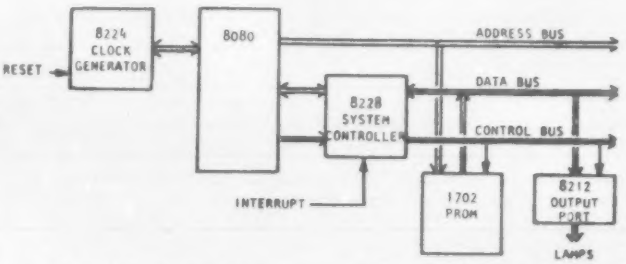


Figure 1 Small 8080 system for labs #2 and #4.

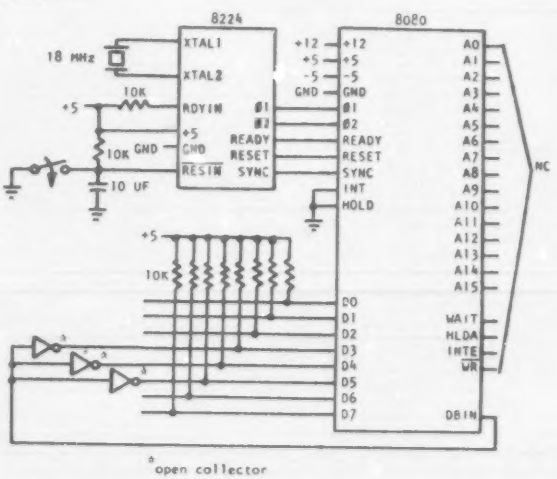


Figure 2 8080 test circuit for lab #3.

Figure 3 Development system.

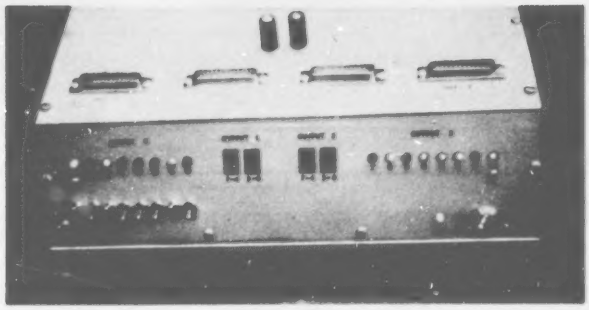


Figure 4 General-purpose I/O box.

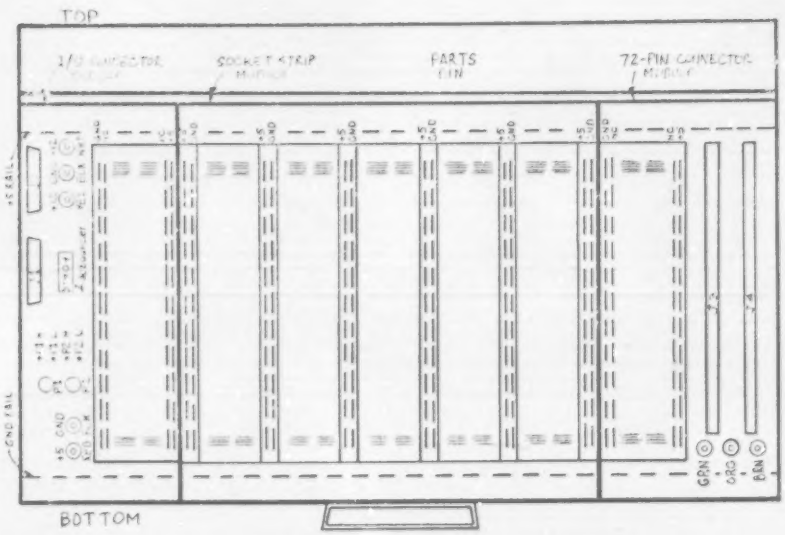


Figure 5 Br[illegible]board layout.

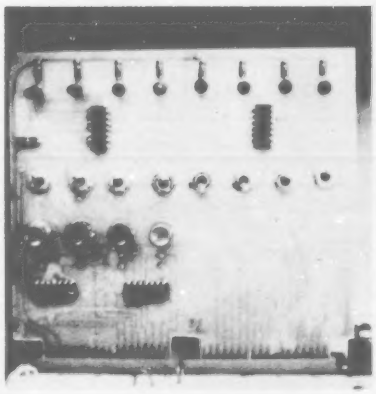


Figure 7 Lamp and switch module.

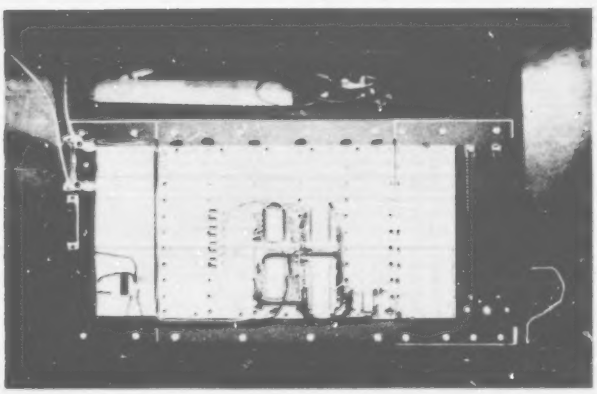


Figure 6 Breadboard kit.

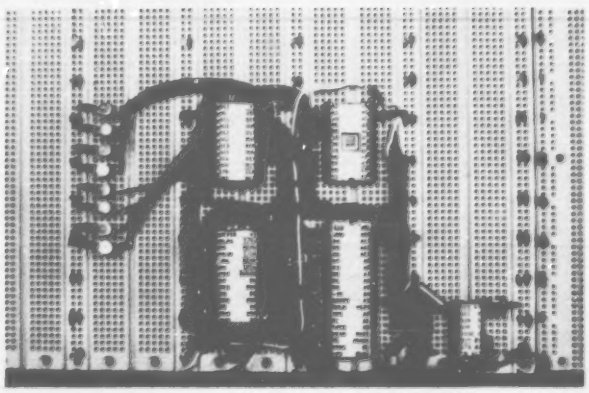


Figure 8 Breadboard wiring for lab #4.

TEACHING MICROCOMPUTER INTERFACING TO NON-ELECTRICAL ENGINEERS

Peter R. Rony and David G. Larsen
Departments of Chemical Engineering and Chemistry
Virginia Polytechnic Institute & State University
Blacksburg, Virginia 24061

ABSTRACT

In view of the fact that the microcomputer will have a profound societal impact in the future, it will become important at most institutions of higher education to teach non-electrical engineers the basic principles and applications of microcomputers. During the past two years, we have developed laboratory hardware and written several self-instructional laboratory workbooks that permit a novice in electronics to learn the basic principles of digital electronics, the 7400-series of integrated circuit chips, asynchronous serial digital communication techniques, microcomputer programming, and 8080A microcomputer interfacing. Within only two quarters, our students can learn sufficient digital electronics and microcomputer interfacing to undertake projects of modest complexity. Student response to the course has been enthusiastic.

With the limited time available to teach the two courses, important compromises have been made in the choice of subjects to be taught to our students. Little time is spent on fundamental concepts of analog electronics, theoretical aspects of digital electronics, integrated circuit manufacturing technology, equivalent logic circuits for flip-flops and MSI circuits, other logic families, and assembly language programming. Once a student develops greater maturity in the practice of electronics, he can decide for himself which of these topics he must acquire to broaden the educational base for his chosen profession.

I. PHILOSOPHY AND RATIONALE

The typical university abounds with non-electrical engineers, and they will be an audience that we must serve as educators if the microcomputer has the societal impact expected of it within ten years. The education of one thousand students per year at a university of 15,000-20,000 students will be common five years from today. For example, all engineers, not just electrical engineers, will require a background in microcomputer interfacing and programming.

During the past two years, we have been experimenting with an approach to the teaching of digital technology--including logic, TTL integrated circuit chips, microcomputer interfacing and programming, and digital communications--to non-electrical engineers. Our audience has consisted of students from a wide variety of disciplines--chemical engineering, chemistry, computer science, psychology, biology, biochemistry, engineering science and mechanics, geology, and mechanical engineering--many of whom have had no prior exposure to electronics. All such students share the common problem that they do not have much course time available during their undergraduate or graduate careers to study electronics

We are not educating our students to become accredited electrical engineers. Rather, we are providing our students with specific training in digital technology that they may find useful professionally. We expect our students to learn how to design, construct, and use digital systems that are not too complex; understand the present and future roles of digital technology; be able to communicate with electrical engineers; and, in general, have a favorable attitude toward digital technology. In short, we are trying to break down the electronics barrier that is commonly encountered by scientists and engineers.

We do not have much course time to fulfill such educational objectives. A typical quarter course consists of two contact hours per week of lecture and three contact hours per week of laboratory. An average quarter lasts for nine weeks. Sixty percent of our students take only one quarter of electronics; thirty percent take two quarters and ten percent take three quarters.

With the exception of our junior ChE course, all of our digital courses are technical electives. Their basic attractiveness to students is determined by our ability to provide a course that students feel is interesting and useful. Some of the educational "experiments" that we have tried or will soon try include the following:

- Teach digital electronics first, and spend essentially no time during the course on basic concepts of analog electronics other than Ohm's law. AC theory, transients, diode behavior, and transistors are discussed only to a very limited extent.
- Eliminate most theoretical concepts--including Boolean algebra, Karnaugh mapping, De Morgan's theorem, and related topics--from the teaching of digital electronics.
- Spend no time on the optimization of digital circuits.
- Eliminate discussions of integrated circuit manufacturing technology and the specific characteristics of transistor circuitry within IC chips.
- Do not dwell long on the equivalent logic circuits for flip-flops or MSI and LSI integrated circuit chips.
- Restrict the course to the TTL logic family, which is popular and inexpensive. Touch only briefly on other modern logic families, such as CMOS and MECL.
- Approach the entire subject of digital electronics from a systems point of view (to a non-electrical engineer, digital components are truly "little black boxes").
- Permit students to breadboard many digital circuits of low or modest complexity rather than several complex digital circuits.
- Integrate the teaching of TTL chips and microcomputers into a single course.

We know that we have done our job when students develop enthusiasm for digital technology and initiate their own plans for continuing education in the field. Such continuing education takes the form of the use of digital techniques in graduate work, the purchase of hobby kits and integrated circuit chips, subscriptions to electronics magazines, and interest in additional courses and seminars on electronics.

We teach the same digital courses to three groups of students: (1) junior chemical engineers who are required by the curriculum to take the "Digital Electronics for Scientists" course, (2) seniors and graduate students who take the "Digital Electronics for Scientists" and microcomputer interfacing courses as electives, and (3) professional scientists and engineers who take our Extension Division short courses in digital electronics and microcomputer interfacing. All three groups respond favorably to our courses, but the professional scientists and engineers remain our most enthusiastic audience.

II. MICROCOMPUTER HARDWARE

A hurdle that we share with many other institutions in the development of digital laboratory courses is the cost of laboratory hardware. We believe that non-electrical engineers deserve more individual laboratory attention from a course instructor than do electrical engineers. We thus have a basic tradeoff: digital hardware vs an instructor's time, i.e., limited amounts of laboratory equipment are available (owing to their cost), we compensate for this limitation by scheduling many different laboratory sections. We have steadily moved toward the opposite tradeoff: plentiful equipment and few scheduled laboratory sessions. At the moment, we have eleven 8080A-based microcomputers and are able to schedule two laboratory sections to handle a total of forty-four students. The microcomputers are portable and inexpensive, and are checked out by the students over the weekend during the latter half of the course. We also use such microcomputers in our Extension Division short courses, both on campus and off.

To minimize costs, we have developed our own hardware, which is described in references 1 through 6. All of our hardware--in digital electronics, microcomputer interfacing, and digital communications--is centered around the very popular solderless breadboards. For experiments with TTL integrated circuit chips, we employ the concept of an *Outboard*®, which is a small printed circuit board that performs a single digital function and connects directly to a solderless breadboard at its outer edges (Figure 1). We have developed Outboards for the basic digital functions, including lamp monitors, logic switches, pulsers, seven-segment displays, latch/displays, clocks, counters, decoders, multiplexers, UARTs, TTL-to-20 mA current loop interfaces, TTL-to-RS 232C interfaces, and an 8-bit microcomputer data bus latch/display. In Figure 1, we show a clock Outboard, lamp monitor Outboard, decoder Outboard, and two latch Outboards, all attached to a solderless breadboard found on one of the microcomputers that we use in our courses. The Outboards simplify the wiring of digital circuits and improve the packing density of digital functions on a solderless breadboard.

Armed with a variety of Outboards, we proceed to use the solderless breadboard approach with all of the microcomputers developed in Blacksburg.[7] For example, the Mark 80 microcomputer contains a solderless breadboard that provides the signals associated with the

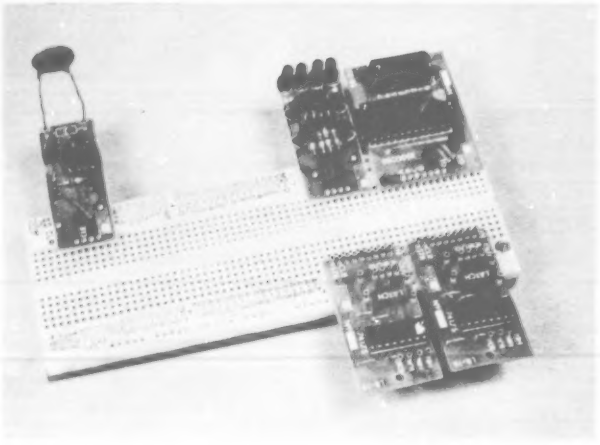


Figure 1. The Mark 80 microcomputer bus socket, shown with several Outboards.

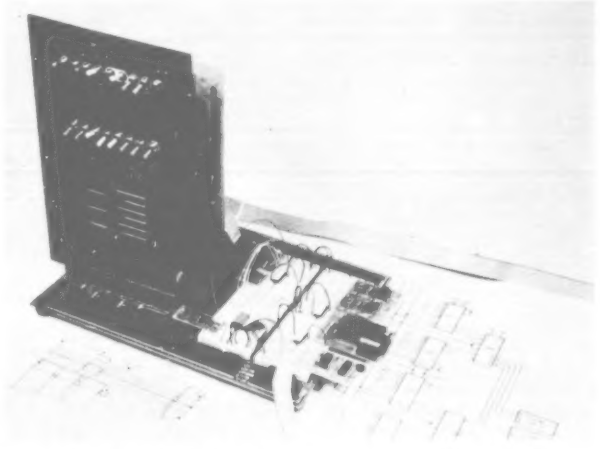


Figure 2. The Mark 80 microcomputer, an 8080-based microcomputer that permits the solderless breadboarding of interface circuits.

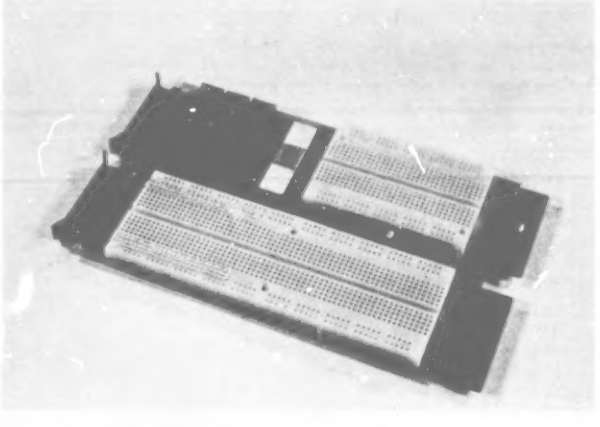


Figure 3. The 8255 programmable peripheral interface board for the Mark 80 microcomputer.

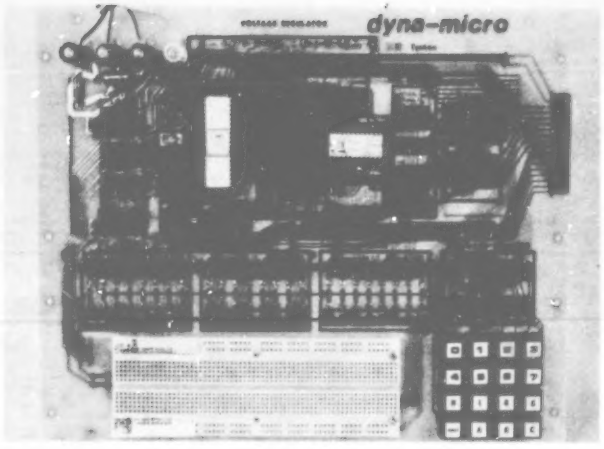


Figure 4. Top view of the Dyna-Micro microcomputer.

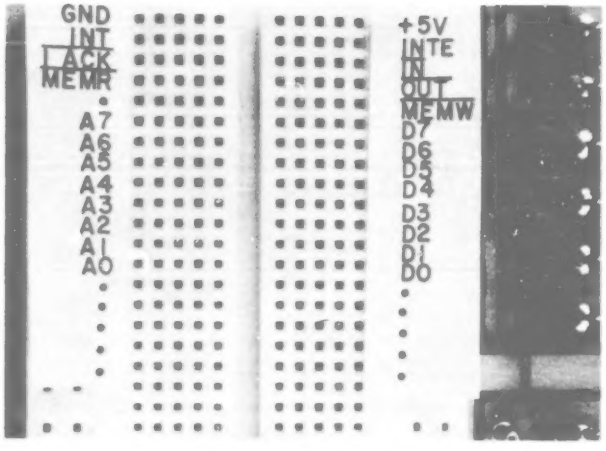


Figure 5. Interface signals available on the Dyna-Micro microcomputer. Also present elsewhere on the board are HALT, WAIT, M1, HLDA, ϕ2, and five decoded I/O channels.

16-bit address bus, the 8-bit bidirectional data bus, the control bus, an 8-bit output port, and eight bits of memory data (Figures 1 and 2). Additional plug-in printed circuit boards, such as the 8255 programmable peripheral interface board (Figure 3), also contain significant breadboarding capability for students. The newer and considerably less expensive Dyna-Micro microcomputer (Figure 4) contains a single solderless breadboard and a limited set of address bus, bidirectional data bus, and control bus breadboard signals (Figure 5).

The use of solderless breadboards, Outboards, and integrated circuit chips permits our students to wire a great variety of interface circuits in our microcomputer laboratory. This is also quite evident in Extension Division hands-on short courses on microcomputer interfacing. One example is the dual microcomputer master-slave handshaking circuit (Figure 6) and associated program developed by a group of electrical engineers in several hours at our recent two-day short course on microcomputer interfacing in Reston, Virginia.[6]

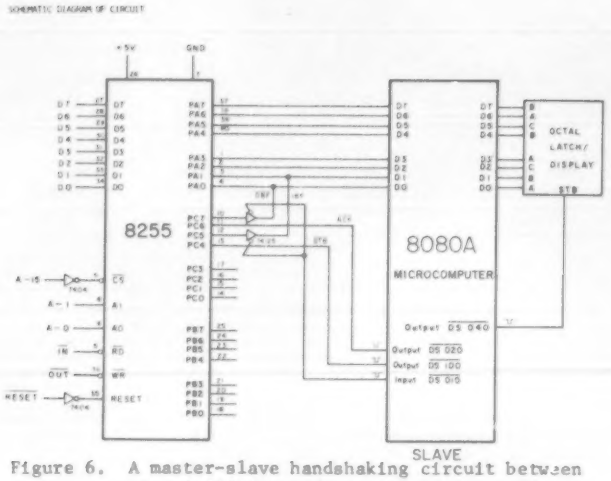


Figure 6. A master-slave handshaking circuit between a pair of 8080A-based microcomputers. The interface consists of an 8255 programmable peripheral interface chip operated in the mode 2 configuration.

III. MICROCOMPUTER EDUCATIONAL SOFTWARE

Another major hurdle associated with the development of a laboratory course on microcomputers is educational software. An instructor usually must develop his own software, i.e., a series of laboratory experiments. Having done so, his product is rarely marketable as a textbook. We have observed that major textbook publishers are not sympathetic to the "limited" markets that they believe are characteristic of laboratory workbooks. Computer manufacturers, with few exceptions, are of no help either. They tend to distribute computer reference manuals rather than self-instructional laboratory workbooks.

We have been very fortunate that a small company that markets solderless breadboards and related products, E&L Instruments, Inc., has had a strong desire to market educational material that demonstrates the use of its products. The result of our university-industry collaboration is a series of books, called *Bugbooks*, that document a laboratory approach to digital technology that we have found to be successful at VPI&SU. The books have been printed in Blacksburg directly from the typewritten originals. We retain the copyrights to the books, have the opportunity to make corrections between printings, and do not have to suffer the whims of major textbook editors and referees, whom we believe stifle the publication of laboratory material. Since our books are distributed nationally, we have had incentive to do a more thorough job with the experiments.

Our fundamental concern has been with the basics of digital technology. Our "basics" do not consist of Boolean algebra, Karnaugh mapping, or digital theory in general, but rather the 7400-series of TTL integrated circuit chips. *We consider the 7400-series of chips to be the ABCs of digital electronics, at least for non-electrical engineers.* We believe that it is important for our students to understand the basic characteristics of the 7400-series of gates, flip-flops, latches, buffers, counters, multiplexers, demultiplexers, decoders, displays, memories, shift registers, and arithmetic/logic units. We have no interest in training our students to construct flip-flops or other MSI circuits from gates and flip-flops. Gates are so inexpensive today that the optimization of gate circuits does not appear to be an important

subject anymore. Our students are not designers of the logic within integrated circuit chips; rather, they are users of chips. Although we do not discuss it, we believe that the transition from TTL to CMOS or I^2L will be relatively easy for most of our students.

Our microcomputer educational software, Bugbooks III and V, treat 7400-series TTL integrated circuit chips as "devices" that can be interfaced to an 8080A-based microcomputer. We do not teach our students how to design microcomputers, nor do we discuss in detail the different available microprocessor chips, such as the 6800, PACE, 9900, SC/MP, etc. We expect our students to be users of existing microcomputers, and therefore focus their attention on the digital communication between a typical microcomputer and the external world. We have chosen the 8080A microprocessor chip because of its great popularity. Two of the largest semiconductor manufacturers, Texas Instruments and National Semiconductor, now second source the 8080 microprocessor chip, as do several other companies. For this reason, we believe that the 8080A is rapidly becoming the industry standard microprocessor and will have a long lifetime.

The use of 7400-series chips as input-output devices has proven to be very productive since (a) our students have already had our course on 7400-series digital chips, (b) such chips are very inexpensive I/O devices, and (c) all the basic principles of microcomputer interfacing can be demonstrated using simple digital circuits constructed from TTL chips. By "basic principles of microcomputer interfacing," we mean the following:

- The generation of sychronization pulses such as input and output device select pulses.
- The output of eight-bit bytes from the 8080A microcomputer to an output device.
- The input of eight-bit bytes into the 8080A microcomputer from an input device.
- Interrupt servicing.
- Writing input-output programs.

In all of our Bugbooks, we train our non-electrical engineers to be able to wire a digital circuit directly from a schematic diagram. Figure 6 is an example of such a diagram. Having wired an interface circuit, the student spends time programming the microcomputer and testing the behavior of the circuit. With simple digital chips, we have tried to make the student understand the function of each pin. With the programmable chips such as the 8251 and 8255, we hope to teach him the basic internal programming options.

IV. MICROCOMPUTER PROGRAMMING

In our first on-campus course on microcomputer interfacing, we did not discuss the 8080A instruction set in enough detail, specially early in the course. The result was that the computer science students did very well, while many other students floundered for the first several weeks. We have read in the electronics literature that electrical engineers have had difficulties learning assembly language programming. Other types of engineers and most physical scientists also experience such difficulties, perhaps more acutely owing to the fact that they are still novices in electronics.

During the following academic year, we plan to provide our students with much more information on microcomputer programming, including detailed programs that illustrate important programming techniques. The Intel 8080 cross-assembler is available at remote job entry stations around campus that are tied to a central IBM 370/168 computer. Most of the computer science students, but few of the engineers and scientists, used it during the spring 1976 academic quarter at VPI&SU. Assembly language programming can be tedious and difficult; students must have ready access to an assembler and should have the opportunity to spend hours at a time programming if they desire.

V. SPECIAL PROJECTS

In the opinion of our students, the most valuable part of our spring 1976 course on microcomputer interfacing was the special project. We do not claim either originality or uniqueness with the special project part of the course. Rather, we would like to document once again the educational principle that students tend to be most motivated when they are permitted to work on a project of their own choice. Our "rules of the game" for the special project were relatively simple:

- The special project accounted for 60% of the course grade.
- Each student was encouraged to select a professor in his own discipline and to work on a project of mutual interest.
- Students could work in teams.
- Both a written report--including microcomputer software, schematic diagrams of hardware, and discussion--and an oral presentation to the entire class were required.
- Microcomputers, integrated circuit chips, power supplies, and Outboards could be borrowed over a weekend.
- Sufficient course time, approximately 45%, was allotted to the special project.
- The final report was made available to the sponsoring professor.

The projects chosen by the students during the spring 1976 quarter included the following titles. Most projects were based upon the use of the Dyna-Micro microcomputer shown in Figure 4.

ASCII-Baudot and Baudot-ASCII conversion using a Teletype Model 15.

Reactor control using a PID algorithm.

Interfacing a microcomputer to a pH meter for the purpose of pH control and data logging.

A digital integrator for NMR spectra (project also included the development of a new cross-assembler for the 8080 instruction set).

Cryptography using asynchronous serial techniques.

An 8080-based rallye computer.

APL graphics display on a 5 × 7 dot matrix.

Synchronous/asynchronous communications test set (SACTS).

Conversion of a single-channel gamma ray spectrograph to a multi-channel gamma ray spectrograph.

Transmission and storage of data taken from rotating machinery.

A data-checking device for a helical wire chamber magnetic spectrometer.

An industrial application of a gamma-ray gross composition measurement.

A printer debugging aid for the HP 21/MX remote station.

Interface circuit between a Biomation 805 transient recorder and the CB² 4004-based data communications link to an IBM 370 computer.

A cassette recorder interface for the Dyna-Micro microcomputer.

Computer interfaced data logger and data transmitter-receiver between a PDP 8/E minicomputer and an 8080-based microcomputer.

The design of an 8080A system to be used for psychological research at VPI&SU.

An 8080A-based microcomputer interfaced to a dissolved oxygen meter.

The above titles illustrate the great variety of student interests, as one would expect. The oral presentations were a valuable part of the course, according to the students who attended them.

VI. LABORATORY COSTS

We currently have eleven 8080A-based microcomputers, thirty complete digital electronics stations, and an excellent assortment of integrated circuit chips. Our costs for this equipment can be summarized as follows:

Thirty breadboarding stations	$ 1800.00
Eleven microcomputers (kit form)	3500.00
Eleven power supplies (from surplus)	200.00
TTL integrated circuit chips	3500.00
Miscellaneous hardware	1000.00
CRT	1200.00

Much of the funding for the above was generated via our Extension Division short course activities.

VII. ADDITIONAL COMMENTS

We have been made aware of the fact that some electrical engineers view our educational techniques and experiments with suspicion. We appear to be violating a number of sacred cows, including the absence of theory from our digital course and the teaching of digital electronics before circuit theory or analog electronics. We repeat our caveats that we are teaching non-electrical engineers, that the course duration is usually short, and that the best that we can hope to achieve is to implant an enthusiastic attitude for electronics and some basic skills in digital techniques. We do believe that digital electronics is becoming easier to learn by novices. When individual gates were expensive, it was necessary to optimize digital circuits. Inexpensive MSI and LSI chips, including microprocessors, have eliminated the need to do so. Inexpensive microcomputers and sophisticated interface chips will shift the emphasis from hardware to software. Finally, programming will become easier as microcomputer-based higher level languages become implemented using inexpensive memory chips.

We believe the mystery of electronics is disappearing, at least to the user of digital integrated circuit chips. In the university, the main problem is one of exposure of the student to the fascinating "new electronics." Such exposure will be necessary, since many more people in the future will need to communicate with others concerning their electronic needs.

VIII. REFERENCES

1. D. G. Larsen and P. R. Rony, *Bugbook I. Logic & Memory Experiments Using TTL Integrated Circuits,"* E&L Instruments, Inc., Derby, Conn., 1974.

2. P. R. Rony and D. G. Larsen, *Bugbook II. Logic & Memory Experiments Using TTL Integrated Circuits,"* E&L Instruments, Inc., Derby, Conn., 1974.

3. D. G. Larsen and P. R. Rony, *Bugbook IIA. Interfacing and Scientific Data Communication Experiments Using the Universal Asynchronous Receiver/Transmitter (UART) and 20 mA Current Loops,"* E&L Instruments, Inc., Derby, Conn., 1975.

4. P. R. Rony, D. G. Larsen, and J. A. Titus, *Bugbook III. Microcomputer Interfacing Experiments Using the Mark 80 Microcomputer, an 8080 System,"* E&L Instruments Inc., Derby, Conn., 1975.

5. P. R. Rony, D. G. Larsen, and J. A. Titus, *Bugbook V. Introductory Experiments in Digital Electronics, 8080A Microcomputer Programming, and 8080A Microcomputer Interfacing,"* E&L Instruments, Inc., Derby, Conn., 1976.

6. P. R. Rony and D. G. Larsen, "Bugbooks and Outboards: A Fresh Approach to Digital Electronics and Breadboarding," WESCON 1975, San Francisco, September, 1975.

7. The Mark 80 ® and Dyna-Micro ® microcomputers were developed by Mr. Jonathan A. Titus, Tychon, Inc., Blacksburg, Virginia 24060.

8. T. Villasenor, W. Homes, R. Krueger, and J. Zenger, private communication.

THE UTILIZATION OF A MICROPROCESSOR BASED COMPUTER SYSTEM IN A TOP-DOWN OPERATING SYSTEM COURSE

Dale H. Grit and Dennis D. Georg
Computer Science Department
Colorado State University
Ft. Collins, Colorado 80523

Summary

An alternative to the abstract modeling or case study approach in the teaching of operating systems is presented. The course which is proposed is organized in a top-down fashion and strongly emphasizes the concept of "learning by doing." Laboratory exercises are suggested for a microprocessor based laboratory environment.

Introduction

The prologue to the COSINE report on the curriculum for operating systems[1] states, "The subject of computer operating systems, if taught at all, is typically a descriptive study of some specific operating system, with little attention being given to emphasizing the relevant basic concepts and principles." The purpose of this paper is to propose a course which would emphasize basic concepts and principles. A top-down approach is used to organize the material in this course. The approach, supported by a microprocessor based computer laboratory, is intended to facilitate the teaching of operating system principles and their applications.

In addition to describing the course content, course organization, and potential supporting laboratory exercises, this paper also briefly considers hardware features desirable for implementing an operating system.

The primary motivation for utilizing a microprocessor based computer laboratory is that such a system is the least expensive mechanism for providing students with hands-on computing. Given such a laboratory, the emphasis of the course will be on "learning by doing." It has been reported by Lane[2] that the "hands-on" approach is an excellent way for students to learn operating system principles. The students are closer to what is actually happening in the systems they design and implement. They can compare the effects of different system designs. The enthusiasm of the students in a "hands-on" environment is hard to generate using other teaching approaches. Other motivating reasons for choosing a microprocessor based laboratory include the modularity of hardware system components, the versatility of module design, and the maintainability of the hardware system by non-engineers.

The top-down course utilizing a microprocessor based environment permits the teaching of the first principles suggested by Denning[3] but avoids both the abstract modeling approach as well as the case study approach. Denning[3], in discussing first principles of systems, points out, "We can no longer afford not to apply the concepts and principles in practice." A primary goal of the top-down approach is to provide students with experience in the applications of first principles.

Most traditional operating systems courses (graduate or undergraduate) are organized in a bottom-up fashion. That is, they start out by examining nucleus functions and I/O in detail and then proceed to memory, process and device management. Such course organization is contained in standard texts on operating systems such as Shaw[4], Brinch Hansen[5], Madnick and Donovan[6], Haberman[7], or Tsichritzis and Bernstein[8]. The top-down approach allows students to do system development on an operational system. This means students have access to debugging facilities, loaders, and other utilities. Students will be required to implement alternative strategies and measure the performance of alternative systems.

Course and Laboratory Description

The basic design of the course is a top-down approach to studying operating systems. Each major section of the course will:

1) Introduce the topic(s) to be studied
2) Discuss the concepts associated with and the implementation of the particular algorithm or strategy used by the current system.
3) Discuss alternative methods and algorithms (mechanisms) to deal with the issue.
4) Evaluate the various alternatives and the trade-offs involved.

The laboratory exercise corresponding to each topic will:

1) Acquaint the student with the current implementation through the use of it.
2) Require the student to implement an alternative strategy.
3) Evaluate the alternative with respect to time, space, coding, effort, necessary hardware support, etc.

It is hoped that, by correlating the lecture topics and laboratory assignments, the student will appreciate the various solutions to operating systems problems and get the satisfaction of implementing a solution and evaluating its effect.

The course outline is contained in Figure 1. The major subdivisions of the course and appropriate parallel laboratory exercises are briefly described in the remainder of this section:

Overview

The introduction to the course is an overview and a high-level discussion of operating systems and the services they provide. This discussion will consider the operating system as a hierarchial collection of software which is used to hide the bare machine. From the viewpoint of the user, the computing machine provides various utilities such as compilers, assemblers, and loaders, etc. which exist on top of the operating system.

The system programmer will usually be programming for a virtual machine which provides low-level routines implementing protection and synchronization mechanisms, I/O programs, scheduling and process primitives, and interprocess communication mechanisms. These low-level routines will be introduced as black boxes which accept certain inputs and yield specific outputs. It will be emphasized that these are the underlying mechanisms used by the system programmer to design and implement policies to improve system performance.

Figure 1. Course Outline

I. Overview of Operating Systems
   A. User Point of View
   B. System Programmer Point of View
      1. I/O Programs
      2. Synchronization Mechanisms
      3. Protection Mechanisms
      4. Process Primitives
      5. Scheduling Primitives
      6. Interprocedure Communication
   C. Description of Laboratory Facilities
      1. Hardware
      2. Software
II. Memory Management
   A. Address and Memory Spaces
   B. Partitioned Allocation
   C. Virtual Memory Systems
      1. Segmentation
      2. Paged
      3. Segmentation with Paging
III. Processor Management
   A. Sequential Processes
   B. Concurrent Processes
   C. Job Scheduling
   D. Process Scheduling
   E. Deadlock
   F. Multiprocessor Systems
IV. Device Management
   A. Techniques for Device Management
   B. Device Characteristics
   C. Channels and Control Units
   D. Buffering
   E. Device Allocation Considerations
   F. Virtual Devices
V. Name Management
   A. File System Concepts
   B. Symbolic File Systems
   C. Logical File Systems
   D. Physical File Systems
   E. Sharing
   F. Protection
VI. Nucleus
   A. Static and Dynamic Linking
   B. Relocatable Loader
   C. Resource Handler
   D. Synchronization Primitives
   E. Command Message Handler
   F. Interrupt Handler
   G. Low Level Input/Output
VII. Conclusions

The laboratory exercises associated with this section of the course will introduce the students to the hardware and software facilities in the laboratory. Students will write programs using available high-level software utilities -- compilers, assembler, text editor, file system, etc. If more than one system configuration exists, the students should become familiar with the alternatives and compare the features that each system provides.

Memory Management

The memory management section of the course will begin by considering the distinction between the address space of a process and the actual locations in memory being referenced. Following the introduction, alternative methods of address mapping will be examined. The first alternative is partitioned allocation, in which the entire job is loaded into a contiguous portion of main memory. Solutions to problems associated with this scheme will be discussed including the user's responsibility for designing overlays.

The second alternative to be considered is a virtual memory organization, the purpose of which is to provide each user with a larger address space than the size of physical memory. Methods of implementing virtual memory to be discussed include segmentation, paging, and a combination of both segmentation and paging.

In conjunction with the lectures on memory management, the students will modify the memory manager of the existing system (e.g., a compaction scheme for partitioned allocation, a garbage collector for a purely segmented system, or an alternative replacement algorithm for a paged system). The students will also be required to write a memory allocation/deallocation routine. After comparing various memory management policies in lecture and in the laboratory, the students will write their own memory manager.

Processor Management

Lecture material for the processor management section will consider alternatives in the management and scheduling of concurrent processes. Performance measures for alternative scheduling policies will also be discussed.

There are numerous laboratory projects which can be associated with this section of the course. Initially, to reinforce the concept of concurrent processes, students will be asked to write a concurrent program. Such a program may do arithmetic computation by creating subprocesses which (conceptually) may execute in parallel.

A comparison of alternative scheduling policies can be accomplished by requiring the students to modify the scheduler in the existing system. Different policies can be compared by utilizing a set of benchmark programs provided by the instructor.

The existing system will include a method for detecting (but not avoiding) deadlock. Students could be given the laboratory assignment of creating and observing the affect of deadlock. Following the lecture material on deadlock avoidance, students may be required to implement one of the avoidance algorithms and observe the resulting system performance.

An allocation scheme for a multiprocessor system may be too difficult to implement (if such a configuration exists on the existing system), so lectures on this topic may have to suffice. The final laboratory project on processor management will require students to replace the processor manager of the existing system with one of their own design.

Device Management

While the students are finishing their processor managers, the lecture will turn to device management. Lectures will present the concept of virtual devices, i.e., the hiding of device dependent characteristics at the lowest possible level. After studying the characteristics of various devices, general techniques for device management will be discussed. In the laboratory, students will implement both channel control routines and buffer pool managers. Low-level I/O programs will be provided until the nucleus is considered. At the conclusion of this section of the course, the devise manager for the existing system will disappear to be replaced by the students' device managers.

Name Management

Throughout the course, the students will have made extensive use of the existing file system. The name management section of the course will deal with the

concepts of file systems in a layered, top-down approach. As part of the symbolic file system, students will implement a directory, including name creation and searching schemes. Students will also be asked to compare alternatives in physical file organization, such as contiguous versus noncontiguous file storage.

The sharing of program and data files will be discussed in lecture and implemented in the laboratory. Students will write programs which run concurrently and use a common subprogram of which there is only a single copy. This exercise will reinforce the concept of reentrancy. The final laboratory exercise in the name management section will require students to implement a read/write/execute protection mechanism for their file systems.

Nucleus

The final section of the course will consider the low-level system utilities upon which the above management schemes have been built. Lecture material, in this section, will discuss various alternatives with respect to these low-level routines and how their implementation will affect the partial systems the students have already implemented. In the laboratory, students will implement routines for static and dynamic linking, relocatable loading and resource handling. Additional laboratory exercises will involve the implementation of synchronization primitives, command message handlers, I/O programs, and interrupt handlers for one or two of the I/O devices.

While the students are finishing the laboratory exercises in this section, the lectures will consider several existing operating systems; the features they provide to users, and the design decisions which were made concerning the issues studied during the course. The lectures will discuss the probable influence of architecture on the design of the systems being examined.

Laboratory Environment

Figure 2 contains an outline of the components of the laboratory. It should be noted that the laboratory environment is flexible in that not all the components are not absolutely necessary. Of course, the available hardware and its limitations will influence the particular design of the laboratory exercises and the software components.

Despite their relatively recent introduction, the alternatives available in a microprocessor based computer system differ substantially. Since neither author is hardware oriented, no attempt will be made to discuss specific hardware features. An excellent introduction to microprocessor and microcomputer hardware technology can be found in reference 9. Hardware features desirable for implementing an operating system for this course include:

1) Cascaded Interrupts
2) Modular Memory Blocks
3) Separate Busses for data, addresses, memory, peripherals, and I/O
4) Multiple Addressing Modes
5) Microprogrammable Chips
6) Clocking Mechanism

The main reason for cascaded interrupts is that the system will need to simultaneously control both fast and slow I/O devices. A clock mechanism is necessary for the implementation of alternative scheduling policies.

Of particular concern when building a microcomputer system is the instruction set of the microprocessor. More flexible instruction sets should have a salutary effect on memory utilization, programming time, and program speed. Microprogramming permits the alteration of the software architecture and such modifications should have an affect on the performance and usability of the system.

Lewis and Siena [10] point out that any I/O device which can be used with a minicomputer can also be used with a microcomputer. However, even though all types of I/O devices can be used, a single microprocessor can only process a limited number of channels [10]. Vacroux [11] discusses microcomputer I/O capabilities.

Figure 2. Laboratory Environment

I. Software
   A. Macro Assembler
   B. Macro Library
   C. Relocatable Loader
   D. Static Linker
   E. Dynamic Linker
   F. Text Editor
   G. Cross Compiler
   H. I/O Routines
   I. Complete Operating System
   J. Cross Assembler
   K. System Check-out Routines
   L. Monitoring Software

II. Hardware
   A. CPU
   B. Memory
   C. I/O
      1. CRT Terminal
      2. Hardcopy Device
      3. Cassette
      4. Floppy Disk
      5. Channel Controllers
   D. Clock Mechanism
   E. Interrupt Mechanism

Conclusion

At this point in time, no specific conclusions regarding the result of this course can be made. However, from the development work completed to date, it appears that such a top-down course will provide students with substantial understanding of the concepts and basic processes of operating systems. They will also see the interaction among the various components of an operating system and the far-ranging effects of certain software and hardware design decisions. The microprocessor laboratory is an economical alternative which will provide students with substantial amounts of hands-on experience. Such experience will be an integral part of this course.

Future papers will report on the results of this course, specific problems encountered and solutions implemented during the teaching of this course, and student reaction to the course organization.

References

1. COSINE Committee Task Force Report, "An Undergraduate Course on Operating System Principles," Computer, 5, 1, (Jan/Feb, 1972), pp. 40-59.

2. Lane, Malcolm G., "A Hands-on Approach to Teaching Systems Programming," SIGCSE Bulletin,7, 1 (February, 1975), pp. 23-30.

3. Denning, P. J., "Operating Systems Principles and Undergraduate Computer Science Curricula," AFIPS Conference Proceedings, Vol. 40, 1972 Spring Joint

Computer Conference, AFIPS Press, Montvale, N.J., 1972.

4. Shaw, A. C., The Logical Design of Operating Systems, Prentice-Hall, Englewood Cliffs, N.J., 1974.

5. Brinch Hansen,P., Operating System Principles, Prentice-Hall, Englewood Cliffs, N. J., 1973.

6. Madnick, S. E., and Donavan, J. J., Operating Systems, McGraw-Hill, New York, N. Y., 1974.

7. Haberman, A. N., Introduction to Operating System Design, Science Research Associates, Chicago, Ill., 1976.

8. Tsichritzis, D. C., and Bernstein, P. A., Operating Systems, Academic Press, New York, 1974.

9. Microprocessors: New Directions for Designers, Edited by Edward A. Torrero, Hayden Book Company, Inc., Rochelle Park, New Jersey, 1975.

10. Lewis, D. R., and Siena, W. L., "Clearing the Interface Hurdles," Microprocessors: New Directions for Designers, Hayden Book Company, Inc., Rochelle Park, New Jersey, 1975.

11. Varcroux, A. G., "Microcomputer I/O Capabilities," Microprocessors: New Directions for Designers. Hayden Book Company, Inc., Rochelle Park, New Jersey, 1975

A STUDENT PROJECT IN IMPLEMENTING AN INTELLIGENT COMPUTER TERMINAL

Michael H. Tindall
Department of Computer Science
Colorado State University
Ft. Collins, Colorado 80523

Abstract

This paper describes a microprocessor-based CRT computer terminal that can be implemented by an undergraduate student in approximately one semester. The project requires the student to utilize a moderate amount of hardware and electronics knowledge, as well as a moderate amount of software programming ability. The goal of the project is to give the student the experience of working on the design of an entire microprocessor-based application, and to provide an integrated hardware/software perspective on microprocessor systems design.

Introduction

The recent availability of low-cost microprocessor chips and supporting components has had a major impact on industrial techniques used for designing many new products. Correspondingly, the need has increased for trained persons capable of effectively utilizing microprocessors and microcomputer systems. In recognition of this need, the Department of Computer Science at Colorado State University is attempting to develop appropriate facilities and laboratories to provide this training.

A goal for the program has been to emphasize the integration of both hardware and software considerations and techniques when working with microprocessor systems. The goal is for our students to be capable of designing and constructing a complete microprocessor-driven system; they must have a moderate knowledge of both electronics and computer hardware and programming techniques and software. Their hardware expertise should include, at a minimum, a thorough understanding of a microcomputer system's bus organization, memory types and utilization, and I/O communications and operations. Their software expertise should include, in addition to familiarity with standard programming techniques, a detailed knowledge of I/O programming and timing considerations, interrupt processing, and software memory management.

The purpose of this paper is to briefly describe a laboratory project that was used as a one semester independent study project for an undergraduate student. The project consisted of designing and implementing a fairly powerful CRT video terminal. The terminal is capable of functioning as a video TTY replacement terminal as well as a moderately sophisticated, cursor-oriented text display system that implements a useful command language[1]. The project was a success at least partly because the desired characteristics of the terminal are easily understood; the student was able to verify from previous experience with commercial CRT terminals that the new terminal did indeed behave properly when the project neared completion. This resulted in a high degree of motivation and enthusiasm from the student. The project was also a success in that the finished project yielded a useful display terminal at fairly low cost.

Terminal Characteristics

The completed terminal will consist of a microprocessor system, a CRT monitor, and an ASCII keyboard. The terminal must be able to be connected to another computer system (host) via a standard serial RS-232 interface.

It is mandatory that the terminal function as a video TTY replacement terminal. The microprocessor system will detect any keypresses from the keyboard and transmit them to the host computer. Similarly, any output characters received by the microprocessor system from the host computer will be displayed on the CRT monitor. Three control characters will be recognized by the microprocessor and implemented: carriage return (CR), line feed (LF), and clear screen. Finally, automatic scrolling of lines on the CRT will be provided by the microprocessor system.

A moderately "intelligent" terminal capability may also be implemented for the project. This capability consists of a more extensive set of control commands that can be issued by the host computer to achieve a fairly sophisticated text editor and text display system. The set of commands basically allow the host computer to dynamically define a number of rectangular boxes or "windows" on the CRT screen, and then to manipulate the information in each window independently of the other windows. A window can be designated as "random" or "scrolling". Each random window has a referenceable cursor associated with it that is controlled by the host computer; all text manipulations specified by the host occur with respect to the current cursor position. Each scrolling window has the characteristic that any output characters or lines from the host computer are automatically scrolled through the screen area designated for that window, without disturbing the rest of the screen.

The "intelligent" terminal command set is adapted from the description given in [1].

Hardware Characteristics

The hardware emphasis for the project is on designing the proper interaction between generally available microcomputer system modules, rather than the detailed electronic designing of the modules themselves. The standard modules we utilized are a microprocessor (Intel 8080A) CPU module, a PROM module, a read/write RAM module, and a serial RS-232 I/O module. These modules, comprising a small MITS Altair 8800 microcomputer system[2], were purchased in kit form for the project; the student was required to properly build the modules and assemble them into a workable microcomputer system. The CRT display was implemented using a special kit (Video Display Module[3]) that plugs onto the system bus and provides a 1K dual port RAM that is directly addressable by the microprocessor CPU (as well as a CRT refresh circuit). Any ASCII characters that the CPU places within the 1K memory will be displayed automatically on the CRT at the appropriate screen location (configured as 16 lines of 64 characters each). This display system has the advantage of being very easy to utilize (with appropriate software) and being

very fast (the complete display can be changed as fast as the CPU can address 1K of memory).

The cost of the components used for the system is approximately $1000 dollars, including an inexpensive keyboard and CRT monitor, but excluding the PROM programming equipment (which was already available at the University). All of the hardware modules are reusable for other projects (or a repeat of the same project) at a later date; however, note that only the first student is required to actually assemble the modules themselves.

Software Considerations

The software for the terminal consists of a set of I/O routines to communicate with the external host computer system (through the serial I/O port, either interrupt driven or by polling the status bits), a simple monitor program to allow the terminal user to select the TTY or "intelligent" terminal mode for the terminal, and a set of subroutines to implement the two different modes (a cross assembler was provided for the student). The TTY mode requires a simple cursor control (the cursor location is saved in a register pair or in the standard RAM), with capabilities to move the cursor location one character position to the right on the screen, detect when the right edge of the screen is encountered, scroll the entire page up one line, and process carriage return, line feed, or clear the entire screen control commands. The "intelligent" mode requires that a data structure be maintained to keep track of the "windows" that are currently defined on the screen. In addition, the more sophisticated control command set must be decoded and each request properly carried out.

Conclusion

A student project to design and implement a video CRT terminal provides a good introduction to microcomputer hardware and software system design. The hardware phase provides experience with a bus architecture and microcomputer system control, using standard functional modules as the basic components. The software phase provides experience in real-time computing, I/O programming, and microcomputer software constraints (such as a small memory and a limited instruction repertoire).

In addition, student motivation and enthusiasm was good, and the finished project provided a useful piece of equipment.

References

1. Irby, Charles H., "Display Techniques for Interactive Text Manipulations", AFIPS Conference Proceedings, Volume 43, 1974 National Computer Conference and Exposition, pp. 247 - 256.

2. Altair 8800 Technical Notebook "The Altair Age -- Altair Documentation", MITS, Inc., 6328 Linn N.E., Albuquerque, NM, 87108.

3. Altair-compatible Video Display Module, Polymorphic Systems, 737 South Kellogg, Goleta, CA, 93017.

AN UNDERGRADUATE DIGITAL SYSTEMS TRACK

Allen L. Larson, Major, USAF
Department of Electrical Engineering
USAF Academy, Colorado

Abstract

The Department of Electrical Engineering at the Air Force Academy is in the process of revising the undergraduate electrical engineering curriculum to include a digital systems track. This paper discusses the historic development which led up to the establishment of this computer track and outlines the rationale behind the courses within the track. It also outlines the lessons learned from teaching digital systems, including microprocessors, to undergraduate students. Finally, the paper discusses the support requirements of a digital systems track.

Historic Background

The Department of Electrical Engineering at the Air Force Academy has been involved in utilizing microprocessors since the fall semester of 1973. A great deal of experience has been accumulated since those initial efforts, and a great many problems have been overcome. The purpose of this paper is to share the results of that effort with others and to stimulate an interchange of ideas beneficial to all concerned educators.

In the fall semester of 1973 a study group was commissioned to study the impact of microprocessors on our electrical engineering curriculum. This study group worked for two semesters, delivered its final report in June 1974. The study group came to several conclusions.

1. Microprocessors represent a quantum jump in technology as significant as the invention of the transistor.

2. The microprocessor would become widely used and knowledge of digital systems based upon microprocessors would be required by graduates of a bachelor's degree program.

3. A primary problem involved in use of a microprocessor would be access cost which was defined to be the sum total of all the training which was required to get a student to the point where he could do meaningful work on a microprocessor.

4. The incorporation of microprocessors into the curriculum should evolve from individual study courses to electrical engineering majors courses and eventually into electrical engineering service courses.

Since the conclusion of this initial study the use of microprocessors by our department has been evolutionary. While the study group was in being an Intel SIM-08 was used by several students in EE 499 (individual study) and EE 464 (design). These initial efforts yielded mixed results. While the students were enthusiastic and learned a great deal, it began to become obvious that even seniors were not prepared to properly use a microprocessor. They simply lacked the proper background in the fundamentals of digital systems.

The fall semester 1974 brought continued utilization of the microprocessor in individual projects and individual study courses. Accompanying this was involvement of the faculty in several research projects utilizing microprocessors. These included the initiation of three projects, (1) an examination of the utilization of a microprocessor for digital filtering and processing of biomedical information, (2) a flexible communications channel error rate monitor, and (3) the development of an automated high-speed data acquisition system. From this research work and student involvement with this research work, many additional lessons were learned relating to teaching microprocessors to students. More beneficial, however, was the expertise gained by the faculty as they conducted their research.

The spring semester of 1974 marked the first utilization of a microprocessor in existing curriculum courses other than individual study and individual project courses. That semester EE 381 (The Digital Computer As A Laboratory Instrument) utilized a much modified Intel SIM-08 to replace the Sigma-5 computer previously used in the course. This course was originally conceived as a single course to teach electrical engineering students the use of a computer for experimental and control work. The Sigma-5 computer was initially used because it existed as part of a hybrid computer laboratory at the Air Force Academy and possessed the necessary analog-to-digital and digital-to-analog conversion capability. From an undergraduate standpoint, however, this system had several drawbacks, the primary of which was access cost. The Sigma-5 computer was simply so powerful and so sophisticated that it took most of the semester to teach the student how to program it. Further, to accomplish many of the functions it was necessary to utilize "canned" Fortran subroutines resulting in the student not thoroughly understanding the operation of the system. Also, due to service maintenance contracts it was impossible to attach external peripherals and laboratory projects directly to the Sigma-5. This made it difficult to illustrate the electrical problems and the programming problems of utilizing a computer which is a component part of an overall larger system.

The use of the modified SIM-08 reduced most of these problems to manageable proportions. The time which had to be spent teaching the language of a computer was greatly reduced due to the fact that the SIM-08 language is far simplier then the language of the Sigma-5. The simple input port/output port structure of the SIM-08 permitted connection of interface circuits directly to the computer and permitted teaching the relationship

between the interface electronics and the programming. The overall reduction in access cost permitted the inclusion, for the first time, of background material about interfacing and peripheral circuits. The simplicity of the system also permitted the introduction, for the first time, of individual student term projects. In these projects, which replace a final examination, the students were to attempt to solve a problem pertaining to the use of a computer as a component of an overall larger system.

The spring semester of 1974 also marked the introduction of a unique training aid called Educomp. This training aid was initially designed for use in EE 381 and was incorporated in an elective portion of an EE service course. Educomp is a basic computer training aid which is coupled with a programmed text which together are designed to teach the student the inner workings of a basic digital computer. This is done by starting with a review of basic logic gates and flip-flops and then showing the student how these items may be logically interconnected to form registers, data selectors, decoders, and memories and then how these larger units may be connected to form an I/O unit, an arithmetic unit, a memory unit, and a control unit. As the student progresses through the programmed text he studies each of these units and their operation and advances to the point where he has assembled the simpliest possible digital computer which will illustrate the principles upon which digital computers are based. The result is 16 instruction, 16 word memory, 4 bit word machine. The student is then taught the process of assembly programming and hand-assembly of his program into machine language and is shown 4 digital and 4 analog applications. For the applications another module is added which includes an A/D and D/A converter. In the final exercises the student processes a 400 Hertz signal with real-time output to an oscillosope.

This training aid proved so effective in EE 381 (with 6 students) that more copies were constructed and it was used with approximately 240 students as an elective minicourse within the second semester Electrical Engineering service course (EE 332). This training aid provided several benefits beyond the material derived by the students. The reaction to the hands-on experience was enthusiastic. Further, many instructors who had not been involved with digital computers came to an understanding of the digital computer as an electronic rather than a software device.

The summer and fall of 1975 marked the involvement in the Department of Electrical Engineering in a solar heating project at the Air Force Academy. As a subteam of the Solar investigation team, members of the department utilized an Intellec microcomputer to implement a data acquisition and heating plant control system in a solar house constructed at the Air Force Academy. This system capitalized upon earlier research and teaching efforts within the department and involved many faculty members who had not previously been involved with microprocessors. The design and installation stage of this project are completed and the project involvement now consists primarily of assisting the solar investigation team with data reduction.

Also in the fall of 1975 the previously mentioned course, EE 381, was moved from the modified SIM-08, which was actually a part of one of the research projects, to an Intel Intellec 8/MOD 80. The conversion from the 8008 to the 8080 microprocessor enabled the student to more easily write programs because of the improvement in the language they were using. Further, the 8008 had been limited in the number of subroutine levels to 7, whereas the 8080 is essentially limited only by memory size.

The significant new development and instructional material in EE 381 was the inclusion of material covering program design. This material was taught as a basic algorithm, called SSD (structured system development). It is a common algorithm which may be used for the development of hardware or software or systems involving both hardware and software. As used in the course it is first taught in this general form and then reformulated to provide an algorithm for the design, implementation, test and evaluation of structured, modular programs. The inclusion of this material has brought a significant improvement in the success of students at writing good programs on a microprocessor.

Also, in the fall of 1974 a department curriculum revision effort was initiated as part of the Air Force Academy's implementation of recommendations made by the Academy 20 Year Study Group. This curriculum revision effort has led to the development of a revised electrical engineering curriculum consisting primarily of four components. The first component consists of the electrical engineering service courses. The second component is made up of the electrical engineering major main courses (electronics) and the remaining two components are the courses involved in the communications track and the digital system track. Initial work in the fall semester of 1975 was to coordinate certain changes with the overall Academy program and to select the course titles and topics for the courses in the new curriculum.

A new training aid, called Edumicro, was also designed in the fall semester of 1975. This training aid is an 8080 based microcomputer which is designed for the undergraduate instructional environment as well as providing a modular system which may be utilized throughout a computer track and also in research projects.

The spring semester of 1976 saw the inclusion of the Edumicro training device into EE 381 and writing of the related programmed text, as well as further refinement of that course. More significant, however, was the continuing effort at design of the new curriculum. The entire department was involved in refinement and a detailing of the purpose, the goals, and the objectives of each course within the new curriculum. The intended purpose of this design effort was to make the new curriculum as efficient as possible. As new material is being included it was essential that no important material was deleted. This curriculum design process is now essentially complete. The computer track (digital systems track) portion of this new curriculum and the rationale behind it are the topics of the remainder of this paper.

Purpose of A Digital Systems Track

Why have a digital systems track in an undergraduate curriculum? The answer to this question is that economics have forced the issue. The advent of the microprocessor has suddenly made it practical to employ a digital computer as a component within larger electronics systems. Basically all electronics systems either process data or control other systems. Microprocessor based electronics systems can easily do both. The only difficulty is that in order to

intelligently use a microprocessor the electrical engineer must understand the fundamentals involved in dedicated digital computer usage. The question is then, what are the fundamentals that he should understand, and can they be taught at the undergraduate level?

The purpose of a digital systems track in an undergraduate curriculum should not be to teach microprocessors as an end in themselves. Instead, it should be to teach the student the fundamentals involved in combining digital hardware and digital software and digital systems theory into an integrated digital system. In this context a microprocessor becomes a vehicle through which the fundamentals involved in dedicated digital systems may be clearly illustrated to the student. The principles taught should be applicable to dedicated computer systems of all sizes. Further, the fundamentals involved in digital systems do not all relate to the microprocessor. The computer is only part of the spectrum of digital systems which must be understood before an engineer can intelligently employ microprocessors.

With this purpose in mind it has been possible to develop a sequence of four courses which form a teachable, supportable, viable, digital systems track within an undergraduate curriculum. Before discussing these courses a caution about a potential pitfall. One of the major pitfalls which microprocessors can lead an educator to is a teaching of what is currently available on the market. Several courses which were reviewed in the process of our curriculum design are merely a discussion of the architectural aspects of microprocessors that are currently on the market. While computer architecture is an interesting topic, the time available in an undergraduate curriculum simply does not justify spending an entire course on it. Further, if specific microprocessors are taught in detail as an end in themselves, the information acquired by the undergraduate will soon be out of date. It is far more important to teach the student the fundamentals involved in digital systems of all varieties and the methods by which he can evaluate the cost/performance tradeoffs of all types of digital systems. One must be very careful not to leap the gap between basic gates and flip-flops and the sophisticated microprocessor.

Digital Systems Track Courses

The digital systems track in our curriculum consists of four courses. These are: (1) EE 380--Modern Logic Design, (2) EE 445--Discrete Signals and Systems, (3) EE 487--Real-time Computation and (4) EE 488--Microprocessor Engineering. These courses do not exist in isolation. They build upon material covered previously. Most significantly, they build upon the first course in electrical engineering which is taken by all students at the Academy as a service course if they are not electrical engineering majors or as the first course in their major, if they are electrical engineering majors. This course, EE 210, is a course teaching classical logic design involving such topics as Boolean algebra, gates, flip-flops, combinatorial logic, state diagrams Karnaugh maps, sequential state machine, etc.

Having this "preprocessor" course ahead of the computer track is very important. It permits inclusion in the computer track of the necessary fundamentals such as modern logic design using medium scale and large scale integrated circuits and advanced state controller design principles including microprogramming. This track also builds upon the student having had the electronics portion of their major prior to involvement in the computer track courses (with the exception of taking the first computer track course concurrently with the last electronics course).

The first course in the computer track is EE 380, Modern Logic Design. The purpose of this course is for the student to acquire an understanding of and a facility with the use of modern large and medium scale integrated circuits involved in a variety of digital hardware systems including hardwired, microprogrammable, and fully programmable systems. It is intended that he will understand the performance costs and complexities of hardware costs in the spectrum between gates and flip-flops and the microprocessor.

The intent then, is that EE 380 be a digital subsystem course concentrating upon the hardware aspects and leading into the programming aspects of digital subsystems. This course builds upon EE 210 which covered the topics traditionally taught in a logic design course. However, the student leaving EE 210 has only an introductory grasp of digital hardware. In EE 380 he must move from this introductory knowledge of digital logic to a clear understanding of more complex digital subsystems.

The course covers as its main topics the dynamic models of logic elements, LSI and MSI building blocks, a review of traditional logic design, modern state control implementation techniques, advanced microprogramming, digital subsystem design problems, and the use of statistics in digital design. The student from EE 210 understands logic and the classical gate and flip-flop minimization techniques. Due to the availability of low-cost complex functions on medium scale and large scale integrated circuits, this knowledge is not entirely adequate for the student. He must progress from the viewpoint that the circuits simply have ones and zeros to an understanding that the circuits are truly electronic devices and relate currents, voltage, and propagation time to the devices. The student must learn to relate the function implemented on an integrated circuit to the electronic input/output specification of that circuit. He must learn the standard building blocks available to him to include multiplexers, decoders, memory units, programmable logic arrays, registers, and counters. From this knowledge of the building blocks he is to progress to the design of state controllers, which are simply hardwired algorithms. He must understand the impact of algorithms upon the design techniques chosen. In EE 380 the student will be taught state controller realization utilizing (1) classical flip-flop techniques, (2) "One-hot" state controllers, (3) counter based controllers, and (4) basic microprogrammed state controllers.

The introduction of the basic microprogrammed state controller and its capitalization upon the read-only memory and programmable logic arrays lead naturally into advanced microprogramming. A method of building state controllers which are microprogrammable and teachable has been developed. This approach starts the student with a register and a read-only memory as the basic microprogrammed state controller. He then progresses through the multiplexer, a command decoder, a subroutine return address register, a loop counter, a data register, and finally an arithmetic logic unit. By progressing in this fashion the student is taught the utilization of medium scale integrated circuits to construct a microprogrammable state controller which approaches a microprocessor in power and complexity. At that

point he may be introduced to the microprocessor as a fully programmable logic system. This approach further clearly illustrates that microprogramming is not the programming of a microprocessor.

Also inherent in the teaching of modern logic design is a discussion of the problems which may be encountered and the use of statistics to estimate mean-time between failures of these complex sub-systems. EE 380 then, is a hardware course which fills the spectrum between the classic logic design course and the microprocessor in a logical, teachable fashion.

EE 445, Discrete Signals and Systems, is intended to give the student an understanding of and a facility with the basic theory involved in analysis of discrete signals and systems and digital signal processing using modern data acquisition control and communications systems. The material reviews the use of Laplace transform methods and continuous time linear system theory using the Z-transform. Instruction includes calculating different equations, transfer functions, impulse responses, and evaluation of the frequency response of discrete systems. This background allows an introductory treatment of digital filtering design as an extension of analog filter design. It also covers the use of discrete Fourier transforms and its implementation by the Fast Fourier Transform. This course was initially taught in the fall semester of 1974 and has been polished to the point where the course is quite successful with undergraduates. The need for this course in a computer track is obvious. The theory involved in employing a digital system to process information derived from an analog source is truly fundamental.

EE 487, Real-Time Computation, is essentially a minor revision of EE 381 discussed previously. The purpose of EE 487 is to teach the design, implementation, documentation, and testing of integrated real-time software and firmware involved in dedicated digital systems with the computer used as a component part of an overall larger system. The student should thoroughly understand the basic inner electronics of digital computers utilizing the aforementioned Educomp training aid; the assembly language programming of a microprocessor system including the use of support systems, program generation, execution and debugging (utilizing Edumicro and the Intellec); the fundamental process of structured system development for the generation of modular reliable systems of software; the architecture and basic principles of operation of computer peripherals to include A/D and D/A converters, solenoids, servo devices, displays, hardcopy devices, sensors, transducers, and mass storage media; the interaction between the software and the hardware involved in the interface, control and communications with such peripherals; and the integration of these topics in a final project. This course has been highly developed and reasonably successful. As an established course its risk factor in the new curriculum is extremely low. This course is also designed as the course which may be taken by Computer Science majors who are interested in dedicated processors and machine language programming.

The final course in the computer track is EE 488, Microprocessor Engineering. The purpose of this course is for the student to understand the fundamental problems of integrating the theory that he has learned in EE 445, the software considerations he has learned in EE 487, and the hardware design he has learned in EE 210 and EE 380 into practical, reliable, and functional microprocessor based digital systems. The primary topics are (1) computer architecture, (2) A/D and D/A subsystems, (3) priority interrupt systems, (4) system interfacing, (5) data communications hardware/software, and (6) system integration problems. This is the course which is designed to teach at a basic level of electronics involved in a complex dedicated system. The material covered is relevant to computer systems of any size and the approach in this course is to use microprocessors (Edumicro) as a vehicle to illustrate the fundamental material which is taught in the classroom. The approach which is planned in this respect is to first use Edumicro as an example of the tradeoffs involved in a computer architecture and to specifically look at the relationship between the CPU timing, the clock frequency, and the memory integrated circuits. The 8080 will be examined as an example of instruction set considerations such as register usage, and addressing.

After this initial examination of a basic Edumicro, the systems principles taught in the course will be illustrated by using this training aid as a processor around which a basic data acquisition system is built. After the student has covered the A/D and D/A subsystems in class he will design and implement in the laboratory a A/D conversion system to meet a set of performance specifications. Following the block on priority interrupt systems and system interfacing, the student will implement in the laboratory a real-time clock, a priority interrupt system, and a simple direct memory access controller to interface his A/D converter subsystem with Edumicro. With this much of the system implemented the student will implement a digital filter. The discussion of the digital filter leads naturally into consideration of tradeoffs such as hardware versus software multiply. The operation of the completed data acquisition system will then serve as a vehicle for the examination of various hardware/software testing and evaluation techniques and an examination of such system problems as cross talk, power supply noise, and system through put.

In summary, these four courses build upon other courses within the new electrical engineering curriculum and give the student a background in the fundamentals of digital hardware, software, theory, and system integration. The topics in these courses have been specifically selected so that they are both teachable and supportable. Two of the courses are already developed. The two new courses are very nearly developed because they build upon work that has been done by the faculty in research projects and in helping students with individual design and individual study projects. This curriculum has been designed based upon nearly three years of experience with microprocessors in undergraduate environment. It is hoped that the discussion of this curriculum and the considerations behind it may be beneficial to other educators as well as the faculty at the Air Force Academy.

AN INTERACTIVE, OPEN ENDED, LOW COST, VISUAL AID TO EDUCATION

Kenneth W. Drake and David Salomon

San Diego State University

San Diego, CA 92182

Abstract

A computer based system is proposed that could serve as a powerful tool in education. The system is basically a combination of a computer, a graphics display, a cassette recorder and a versatile, multipurpose keyboard. It is unique among similar existing systems because of the following features:

1. Man-Machine Interaction. Using the system is not just a passive watching of the display. The user can interact with the computer through the keyboard. He can press keys and manipulate controls to enter data into the computer, answer questions put by it and control the display. This feature makes the learning process much more interesting and stimulating. It develops creativity, experimentation and curiosity. A child who plays "etch a sketch" with the system in preschool, is taught spelling by it in first grade, and is drilled by it in arithmetic during his second grade, would eventually get curious about it, ask how it works and may try to experiment with it and develop new applications for it.

2. Open Endedness. The system is driven by a computer and each application would be defined by a program. These programs would be recorded on individual cassettes and the user would use a regular cassette recorder in order to load the program into the computer. This means that a library of programs could be developed where programs would be distributed and exchanged among users, offering an unlimited variety of applications in a single system.

3. Low Cost. This is an important feature of the system that would make it affordable even by small schools. This feature is achieved by using a microprocessor as the computing unit and readily available home-products devices (such as a TV set and a cassette recorder) instead of expensive commercial computer peripherals.

General Description

Traditionally, microprocessors have mostly been used in control applications because of their limited computational ability. With the recent advent of 16-bit microprocessors, however, it has become possible to buy a central processing unit with the computing power of a small minicomputer for less than $200.00.

Commercial displays are designed to display information directly from a computer, are fast, easy to use but expensive. A regular TV set, on the other hand, can also display information and is readily available. Since it is not designed to display from a computer, a special interface is needed to translate information stored in a computer to TV signals. Such interfaces are already used with many small computers. Their only disadvantage is slow speed. Only slowly varying pictures could therefore be displayed, which happens to be enough for most educational applications.

Using a cassette recorder is also an important cost saving feature. It is already a popular and recognized technique which expands the system's capabilities in two ways. First, programs could easily be loaded from the cassette. Second, the same cassette could include audio messages that can be played, under program control, during program execution, thus providing feedback, instructions, and rewards from the computer.

Technical Description

Multiprocessing would be used with a 16-bit microprocessor[1-3] as the control processing unit and two 8-bit microprocessors as the input and output processors. The input processor would be driven by programs stored in a ROM. Its main tasks would be monitoring the keyboard for user commands and executing those commands. They may direct the input processor to load an application program from the cassette to RAM, to reset the CPU and to interrupt it.

The keyboard will be a combination of an ASCII keyboard, potentiometers and joysticks and will be blank. For each application, a template will be mounted on it with names for the various keys.

Using a cassette recorder to record digital information is a popular method used expecially with small, low cost computer systems.[4,5] Methods have been developed to obtain reliable recording and retrieval even with inexpensive recorders[6-8] including the use of error-correcting codes.[9]

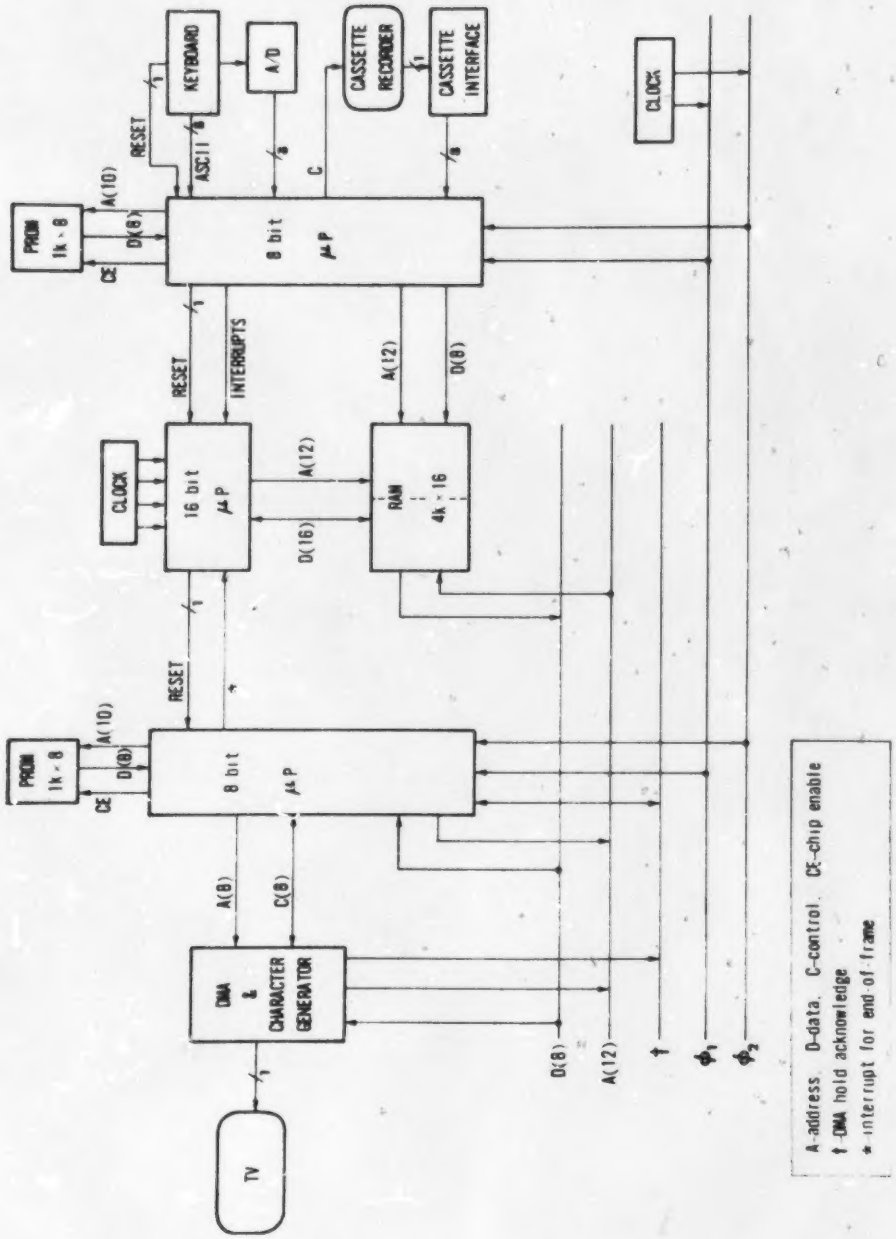
The output processor, again driven by a program in RAM, would be used to control the TV interface. Interfaces to regular TV sets are typically of two types, a character generator[10] that displays characters from a given vocabulary (stored in a ROM), and a dot generator[11] that displays dots at any desired point on the screen (with different intensities or colors). The interface proposed here is a combination of both. It can display dots and also (in separate raster scans) characters whose codes are stored as dot matrices in a ROM.

The output processor will send the following information to the interface.

1. Turn the display on/off.
2. Display in color or B/W.
3. Intensity.
4. Mode of display (dots or characters)
5. Beginning address in RAM. This would be used by the interface to perform a DMA from RAM in either a dot or a character mode. At the end of each raster scan, the TV interface sends back a control signal to the output processor, which in turn interrupts the CPU.

The CPU executes an application program in RAM. The program sets display information in RAM and can be interrupted by the user from the keyboard.

The general design of the system is shown below. Note especially that each of the 8-bit microprocessors can only access half of the existing RAM. This half, which originally is organized as 2K x 16, appears to it as a 4K x 8 and, therefore, 12 address lines are needed between RAM and the I/O processors.



Possible Applications of the System

Shape and color recognition.

A preschool child is asked to copy a displayed shape (using special controls such as joysticks), to recognize a shape (by pressing a key labeled with that shape) or to mix a certain color (using three controls that change the color mixture on the display).

Teaching elementary reading.

Any size letters could be displayed on the screen, combined into words, moved on the screen, erased, etc.

Teaching spelling.

The computer displays a few words and pronounces each. It then asks the student to spell one of those words using the keys. Rewarding can be done by an audio message and the computer maintains a score.

Teaching elementary arithmetic.

Various arithmetic operations can be displayed step by step with concepts such as carry and borrow visually demonstrated.

Develop readiness skills.

A dot moves randomly on the screen and the student must follow it using a joystick. The computer increases the speed of the dot and can also maintain a score. A two-dot version can be used as a two-player game.

Teaching geometry and calculus.

Mathematical functions such as curves can be displayed and varied slowly to demonstrate their dependence on various parameters.

Teaching computer programming.

The process of executing a program can be simulated by the system. Each statement is displayed during its simulation, together with register contents, error messages, etc.

The execution of various algorithms could be animated. Searching and sorting, for example, could be demonstrated by displaying the animated representations of the arrays being searched or sorted.

Summary

The proposed system is an educational tool that can be used to set up stimulating demonstrations and experiments in a wide range of educational areas. The combination of interaction, open-endedness, audio-video output and low cost is unique and offers virtually unlimited educational possibilities. All of the required technologies currently exist and systems definition and implementation for this application area are the major tasks.

Bibliography

1. TMS-9900 Microprocessor Data Manual. Texas Instruments, 1975.
2. PACE Technical Description, National Semiconductor, Inc., 1975.
3. CP 1600 Users Manual. General Instrument Corp., 1975.
4. Micro Sphere 200. The Sphere Corp., Bountiful, Utah.
5. Micro-68. EPA Inc., San Diego, California.
6. E. Wolf, "Ratio Recording for Lower Cassette Recorder Cost," Computer Design (December 1972) p. 76.
7. D. Lancaster, "Serial Interface," BYTE (September 1975) p. 29.
8. BYTE, the March 1976 issue.
9. Soo Nam Kim, "An Inexpensive Audio Cassette Recorder Interface for Microprocessors," EDN (March 1976) p. 83.
10. D. Lancaster, "TV Typewriter Cookbook," Sams Publishing, Indianapolis, Indiana, 1976 (to be published).
11. T. Walker, "The TV Dazzler," Popular Electronics (February 1976) p. 31.